Anatol GREMALSCHI
Iurie MOCANU
Ion SPINEI

INFORMATICA

Limbajul PASCAL



Manual pentru clasele IX-XI

A. Gremalschi, I. Mocanu, I.Spinei

Informatica

Limbajul PASCAL

Manual pentru clasele IX-XI

Chişinău, Î.E.P. *Ştiinţa*
1999

CZU 004 (075.3)

Aprobat pentru editare de Consiliul Ministerului Educației și Științei al Republicii Moldova (nr. 1/4.1 din 26 august 1998)

Redactor: *E. Grosu*
Tehnoredactor: *N. Duduciuc*
Corectori: *M. Belenciuc, E. Pistrui*

Machetare computerizată: *Î.E.P. Știința*

Coli editoriale 14,0. Coli de tipar conv. 16,0
Comanda nr 1115

Întreprinderea Editorial-Poligrafică ***Știința***
str. Academiei, 3; MD 2028, Chișinău, Republica Moldova
Tel. (3732) 73-96-16; 73-99-83. Fax (3732) 73-96-27

Firma Editorial-Poligrafică *Tipografia Centrală*
str. Florilor, 1, or. Chișinău, MD 2068, Republica Moldova
Departamentul Activități Editoriale,
Poligrafie și Aprovizionare cu Cărți

ISBN 9975-67-103-9

Cuprins

INTRODUCERE

Limbajul de programare PASCAL a fost elaborat în anul 1970 de profesorul Niklaus Wirth de la Universitatea Tehnică Zürich (Elveţia). Numit astfel în cinstea matematicianului şi filosofului francez Blaise Pascal, limbajul a cunoscut o răspîndire mondială şi a fost implementat practic pe toate categoriile de calculatoare: supercalculatoare, calculatoarele mari şi calculatoarele personale. Pe parcursul anilor elementele limbajului standard PASCAL au fost îmbogăţite substanţial, creîndu-se astfel versiunile Turbo PASCAL 6.0, Turbo PASCAL 7.0, Borland PASCAL şi chiar limbaje noi, de exemplu, MODULA-2, DELPHI ş.a.

Manualul de faţă a fost elaborat în conformitate cu Programa de informatică, adoptată de Ministerul Educaţiei şi Ştiinţei al Republicii Moldova şi are drept scop însuşirea de către cititor a cunoştinţelor necesare pentru elaborarea şi depanarea programelor PASCAL.

Capitolul 1 include expunerea unor cunoştinţe fundamentale din teoria limbajelor de programare: metalimbajul BNF şi diagramele sintactice. În continuare este prezentat alfabetul şi vocabularul limbajului PASCAL: simbolurile speciale şi cuvintele-cheie, identificatorii, numerele, şirurile de caractere, etichetele, directivele şi separatorii.

În capitolul 2 sînt expuse tipurile de date întregi, reale, booleene, caracteriale, enumerare şi subdomeniu. Tot în acest capitol se studiază mijloacele de definire a tipurilor de date simple, declaraţiile de variabile şi definiţiile de constante.

Capitolul 3 este consacrat conceptului de acţiune şi conţine descrierea instrucţiunilor simple şi instrucţiunilor structurate ale limbajului. O atenţie deosebită se acordă studierii sintaxei şi metodelor de evaluare a expresiilor PASCAL, modului de organizare a proceselor de calcul cu ajutorul instrucţiunilor structurate. Tot în acest capitol sînt incluse generalităţi despre structura unui program PASCAL.

Capitolul 4 include un amplu material teoretic şi practic referitor la definirea şi prelucrarea datelor structurate: tablourilor, şirurilor de caractere, articolelor, mulţimilor şi fişierelor. Sînt prezentate metodele de creare şi prelucrare a fişierelor: asocierea fişierelor PASCAL cu fişiere externe, scrierea şi citirea componentelor unui fişier.

Funcţiile şi procedurile limbajului PASCAL se studiază în capitolul 5. Sînt prezentate în detalii mecanismele de comunicare prin parametri-valoare, parametri-variabilă, parametri-funcţie/procedură şi variabile globale. Tot în acest capitol se studiază subprogramele recursive.

În capitolul 6 se studiază tipul referinţă, variabilele dinamice şi structurile de date create pe baza lor: liste, stive, cozi, arbori. Sînt examinate operaţiile uzuale efectuate asupra structurilor dinamice de date şi problemele legate de aplicarea iteraţiei şi recursiei.

Metodele de elaborare a produselor program sînt expuse în capitolul 7. Sînt prezentate elemente de programare modulară şi programare structurată, metode de testare şi depanare a programelor.

Materialul este expus în aşa mod, încît sintaxa şi semantica limbajului PASCAL să fie înţelese şi însuşite metodic, de la instrucţiuni şi tipuri de date simple la structuri dinamice de date, programe, proceduri şi funcţii. Fiecare porţiune de materie teoretică este urmată de un şir de exemple, exerciţii şi întrebări de control. Se consideră că cititorul va introduce şi va lansa în execuţie programele propuse, va răspunde la întrebări şi, în caz de necesitate, va elabora programele respective.

Toate programele incluse în manual au fost testate şi depanate în mediul de programare Turbo PASCAL 7.0.

Capitolul 1

VOCABULARUL ȘI SINTAXA LIMBAJULUI

1.1. INIȚIERE ÎN LIMBAJUL PASCAL

Vom examina următorul program:

```
1  Program P1;
2   { Suma numerelor întregi x,y,z }
3  var x, y, z, s : integer;
4  begin
5    writeln('Introduceți numerele întregi x,y,z:');
6    readln(x, y, z);
7    s:=x+y+z;
8    writeln('Suma numerelor introduse:');
9    writeln(s);
10 end.
```

Numerele *1, 2, 3, ..., 10* din partea stîngă a paginii nu fac parte din programul PASCAL. Ele servesc doar pentru referirea liniilor în explicațiile ce urmează.

Linia 1. Cuvîntul **program** este un cuvînt rezervat al limbajului, iar P1 este un cuvînt utilizator. Cuvintele rezervate servesc pentru perfectarea programelor, iar cuvintele utilizator — pentru denumirea variabilelor, subalgoritmilor, programelor, constantelor ș. a.

Linia 2. Este un text explicativ, un comentariu. Comentariul începe cu simbolul „{" și se termină cu „}". Comentariul nu influențează în nici un fel derularea programului și este destinat exclusiv utilizatorului.

Linia 3. Cuvîntul rezervat **var** (*variable* — variabilă) descrie variabilele x, y, z și s, utilizate în program. Cuvîntul integer (*întreg*) indică tipul variabilelor respective. Prin urmare, x, y, z și s pot avea ca valori numai numere întregi. Linia în studiu formează partea declarativă a programului.

Linia 4. Cuvîntul rezervat **begin** (început) indică începutul părții executabile a programului.

Linia 5. Afişarea unui mesaj la dispozitivul-standard de ieşire, în mod obişnuit pe ecran. Cuvîntul `writeln` (*write line* — scrie şi trece la linie nouă) reprezintă apelul unui subalgoritm-standard, argumentul fiind textul mesajului ce se afişează:

```
Introduceţi numerele întregi x, y, z:
```

Menţionăm că apostrofurile nu fac parte din textul care va fi afişat.

Linia 6. Citirea a trei numere de la dispozitivul-standard de intrare, în mod obişnuit — tastatura. Numerele sînt tastate în aceeaşi linie şi sînt despărţite de unul sau mai multe spaţii. După tastarea ultimului număr se acţionează tasta `<ENTER>`. Numerele citite sînt depuse în variabilele `x`, `y`, `z`. Cuvîntul `readln` (*read line* — citire şi trecere la linie nouă) reprezintă apelul unui subalgoritm-standard. Argumentele subalgoritmului sînt numele variabilelor în care sînt memorate numerele întregi introduse.

Linia 7. Instrucţiunea de atribuire. Variabila `s` primeşte valoarea `x + y + z`.

Linia 8. Afişarea mesajului

```
Suma numerelor introduse:
```

la dispozitivul-standard de ieşire.

Linia 9. Afişarea valorii variabilei `s` la dispozitivul-standard de ieşire.

Linia 10. Cuvîntul rezervat **end** indică sfîrşitul părţii executabile, iar punctul — sfîrşitul programului.

Prin urmare, un program în limbajul PASCAL este alcătuit din următoarele componente:

- ✓ antetul, în care se specifică denumirea programului;
- ✓ partea declarativă, în care se descriu variabilele, funcţiile, subalgoritmii etc. folosiţi în program;
- ✓ partea executabilă, care include instrucţiunile ce urmează să fie executate într-o anumită ordine de calculator.

Întrebări şi exerciţii

❶ Introduceţi şi lansaţi în execuţie programul `P1`.

❷ Care sînt părţile componente ale unui program PASCAL?

❸ Introduceţi şi lansaţi în execuţie următorul program:

```
Program P2;
  { Afişarea constantei predefinite MaxInt }
 begin
  writeln('MaxInt=', MaxInt);
 end.
```

❹ Indicaţi antetul, partea declarativă şi partea executabilă ale programului P2. Explicaţi destinaţia fiecărei linii a programului în studiu.

❺ Elaboraţi un program care afişează pe ecran pătratul şi cubul numărului întreg introdus de la tastatură.

1.2. METALIMBAJUL BNF

Un limbaj de programare se defineşte prin sintaxa şi semantica lui. E cunoscut faptul că sintaxa este un set de reguli care guvernează alcătuirea propoziţiilor, iar semantica este un set de reguli care determină înţelesul, semnificaţia propoziţiilor respective. În cazul limbajelor de programare, echivalentul *propoziţiei* este *programul.*

Evident, sintaxa oricărui limbaj de programare poate fi descrisă cu ajutorul unui limbaj de comunicare între oameni, de exemplu româna, engleza, franceza etc. S-a considerat însă că o astfel de descriere este voluminoasă şi neunivocă. Pentru o descriere concisă şi exactă a sintaxei limbajelor de programare s-au elaborat limbajele speciale, denumite *metalimbaje.* Cel mai răspîndit metalimbaj este cunoscut sub denumirea de *BNF — Forma Normală a lui Backus.*

Metalimbajul BNF utilizează următoarele simboluri:

simbolurile terminale, adică simbolurile care apar exact la fel şi în programele PASCAL;

simbolurile neterminale, care desemnează unităţile (construcţiile) gramaticale ale limbajului.

Simbolurile neterminale se înscriu între semnele „<" şi „>".

De exemplu, cifrele 0, 1, 2, ..., 9, literele A, B, C, ..., Z sînt simboluri terminale, iar *<Cifră>, <Literă>* sînt simboluri neterminale.

Descrierea sintaxei limbajului PASCAL constă dintr-un set de formule metalingvistice.

Prin **formulă metalingvistică** vom înţelege o construcţie formată din două părţi, stînga şi dreapta, separate prin simbolurile „::=" ce au semnificaţia de *„egal prin definiţie".* În partea stîngă a formulei se găseşte un simbol neterminal.

O formulă metalingvistică permite descrierea, în partea ei dreaptă, a tuturor alternativelor posibile de definire a simbolului neterminal, prin folosirea caracterului „|" cu semnificaţia *„sau".*

De exemplu, formula

<Cifră> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

defineşte unitatea gramaticală *<Cifră>* ca fiind unul din caracterele (simbolurile terminale) 0, 1, ..., 9.

La fel se interpretează şi formula metalingvistică:

<Literă> := a | b | c | d | e | f | g |
h | i | j | k | l | m | n |
o | p | q | r | s | t | u |
v | w | x | y | z

În partea dreaptă a unei formule metalingvistice pot apărea două şi mai multe simboluri consecutive. Situaţia corespunde operaţiei de **concatenare** (alipire) a lor.

Astfel:

<Id> ::= *<Literă>* | *<Literă><Cifră>*

defineşte construcţia gramaticală *<Id>* ca fiind o literă sau o literă urmată de o cifră.

Exemple: a, a1, c3, x4, d, e8.

În unele situaţii alternativele de definire a unui simbol neterminal se pot repeta de un număr oarecare de ori (chiar de zero ori), fapt ce va fi marcat prin încadrarea lor în acoladele {, }.

De exemplu, formula

<Întreg fără semn> ::= *<Cifră>* {*<Cifră>*}

defineşte simbolul neterminal *<Întreg fără semn>* ca o secvenţă nevidă de cifre. Secvenţele 0, 0000, 001, 1900, 35910 sînt conform acestei definiţii, iar secvenţa 3a5910 — nu.

Formula

<Identificator> ::= *<Literă>* {*<Literă>* | *<Cifră>*}

are următoarea semnificaţie: un identificator începe cu o literă; după această literă poate urma o secvenţă finită de litere sau cifre. Astfel, a, a1, a1b, a23x, a14bxz sînt conforme cu această definiţie, dar 2a — nu.

În cazul în care alternativele de definire a unui neterminal sînt opţionale (pot lipsi), ele se încadrează în parantezele drepte [,].

De exemplu, formula

<Factor scală> ::= [+ | -] *<Întreg fără semn>*

defineşte factorul de scală ca un număr întreg fără semn care poate fi precedat de + sau -. Astfel, 1, +1, -1, 20, +20, -20, +003 sînt conforme cu această definiţie, dar 3-5 — nu.

Atragem atenţia că simbolurile [,], {, } aparţin metalimbajului şi nu trebuie confundate cu simbolurile corespunzătoare utilizate în programul PASCAL.

Întrebări şi exerciţii

❶ Explicaţi termenii *sintaxă* şi *semantică*.

❷ Care este destinaţia unui metalimbaj?

❸ Cum se defineşte sintaxa unui limbaj cu ajutorul metalimbajului BNF?

❹ Sintaxa unui limbaj foarte simplu este descrisă cu ajutorul următoarelor formule metalingvistice:

<*Cifră*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<*Număr*> ::= <*Cifră*> {<*Cifră*>}
<*Semn*> ::= + | -
<*Expresie aritmetică*> ::= <*Număr*> {<*Semn*><*Număr*>}

Care din expresiile aritmetice ce urmează sînt numere?

0	0+0	0000
1	11100+1	0001
11100	11100–1	–152
00011	931	+351
20013	614	412

Care din expresiile aritmetice ce urmează sînt corecte?

0+1	-13	21+00000
1+0-3	21+-16	39+00001
0+0+4	-21-16	00001-00001
1+1-9	68-13	379-486
6+6+21	42+650	31+12-51+861

❺ Sintaxa unui limbaj de comunicare utilizator-calculator este definită după cum urmează:

<*Disc*> ::= A: | B: | C: | D: | E:
<*Listă parametri*> ::= <*Disc*> {, <*Disc*>}
<*Nume comandă*> ::= Citire | Copiere | Formatare
<*Comandă*> ::= <*Nume comandă*><*Listă parametri*>

Indicaţi comenzile corecte sintactic:

a) Citire	f) Copiere A: B:
b) Citire A:	g) Citire D
c) Copiere F:	h) Formatare D:,F:
d) Copiere A:,	i) Copiere E:,A:,
e) Formatare D:,E:	j) Copiere F:,A:

1.3. DIAGRAME SINTACTICE

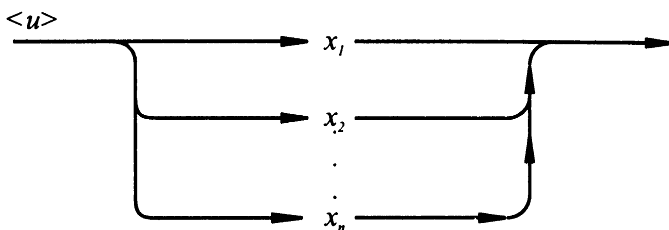
Diagramele sintactice descriu mai clar sintaxa unui limbaj de programare. Reprezentarea prin diagrame poate fi derivată din notaţia BNF, după cum urmează.

Fiecărui simbol terminal îi corespunde un cerc sau un oval în care se înscrie simbolul respectiv. Simbolurile neterminale se înscriu în dreptunghiuri. Ovalurile şi dreptunghiurile se reunesc conform diagramelor din *fig. 1.1*.

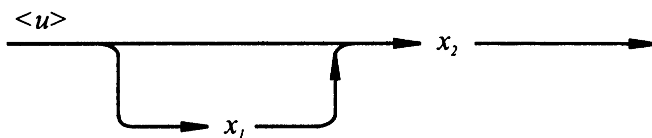
Concatenare: : $<u> ::= x_1x_2...x_n$



Alternare: $<u> ::= x_1|x_2|...|x_n$



Prezenţă opţională: $<u> ::= [x_1]x_2$



Repetare: $<u> ::= \{x_1\}x_2$

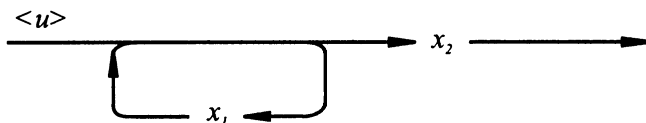


Fig. 1.1. Reprezentarea formulelor *BNF* prin diagrame sintactice

În *fig. 1.2* sînt prezentate diagramele sintactice pentru unităţile gramaticale *<Întreg fără semn>*, *<Identificator>* şi *<Factor scală>*, definite în paragraful precedent. Se observă că fiecărui drum în diagrama sintactică îi corespunde o secvenţă de simboluri terminale corectă sintactic.

Întrebări şi exerciţii

❶ Care este destinaţia diagramelor sintactice?

❷ Cum se reprezintă simbolurile terminale şi simbolurile neterminale pe diagramele sintactice?

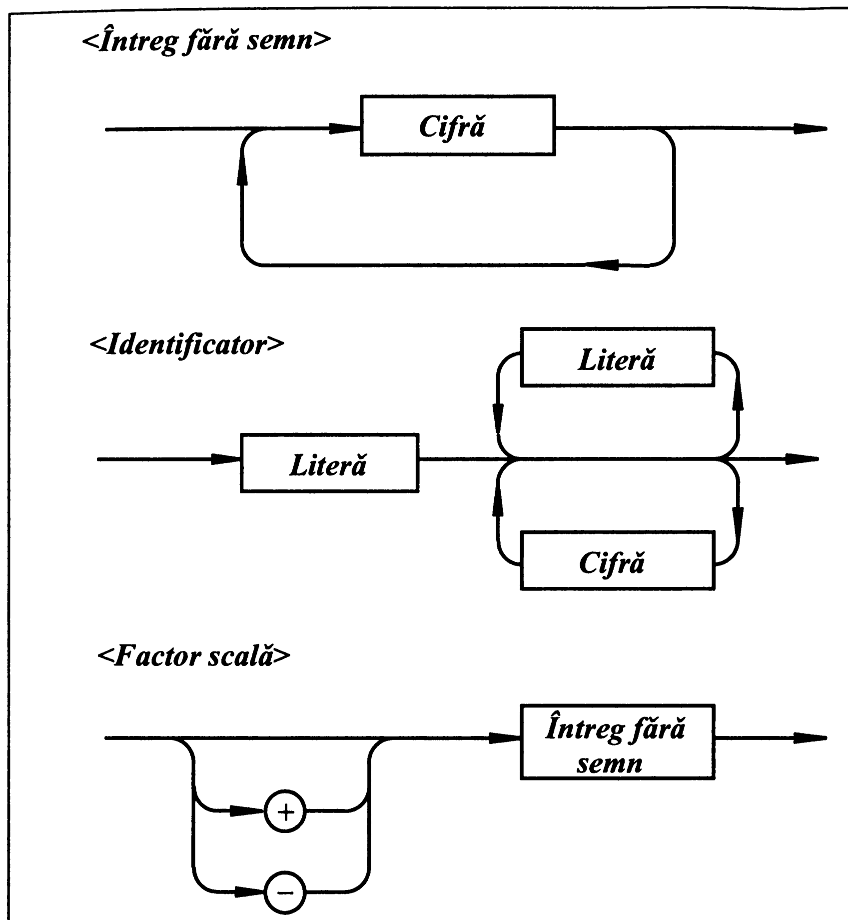


Fig. 1.2. Diagramele sintactice *<Întreg fără semn>*, *<Identificator>* şi *<Factor scală>*

❸ Cum se reprezintă formulele BNF pe diagramele sintactice?

❹ Reprezentaţi cu ajutorul diagramelor sintactice:

<Cifră> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Număr> :: = *<Cifră>* {*<Cifră>*}
<Semn> ::= + | -
<Expresie aritmetică> ::= *<Număr>* {*<Semn><Număr>*}

❺ Reprezentaţi cu ajutorul diagramelor sintactice:

<Disc> ::= `A:` | `B:` | `C:` | `D:` | `E:`
<Listă parametri> ::= *<Disc>* {, *<Disc>*}
<Nume comandă> ::= `Citire` | `Copiere` | `Formatare`
<Comandă> ::= *<Nume comandă><Listă parametri>*

❻ În *fig. 1.3* sînt prezentate diagramele sintactice care definesc unitatea gramaticală <*Număr fracționar*>. Determinați care din secvențele ce urmează sînt conforme acestor diagrame:

0.1	.538
+0.1	721.386.
-0.0	-421
9.000	247.532
-538.	+109.000

❼ Scrieți formulele BNF care corespund diagramelor sintactice din *fig. 1.3*.

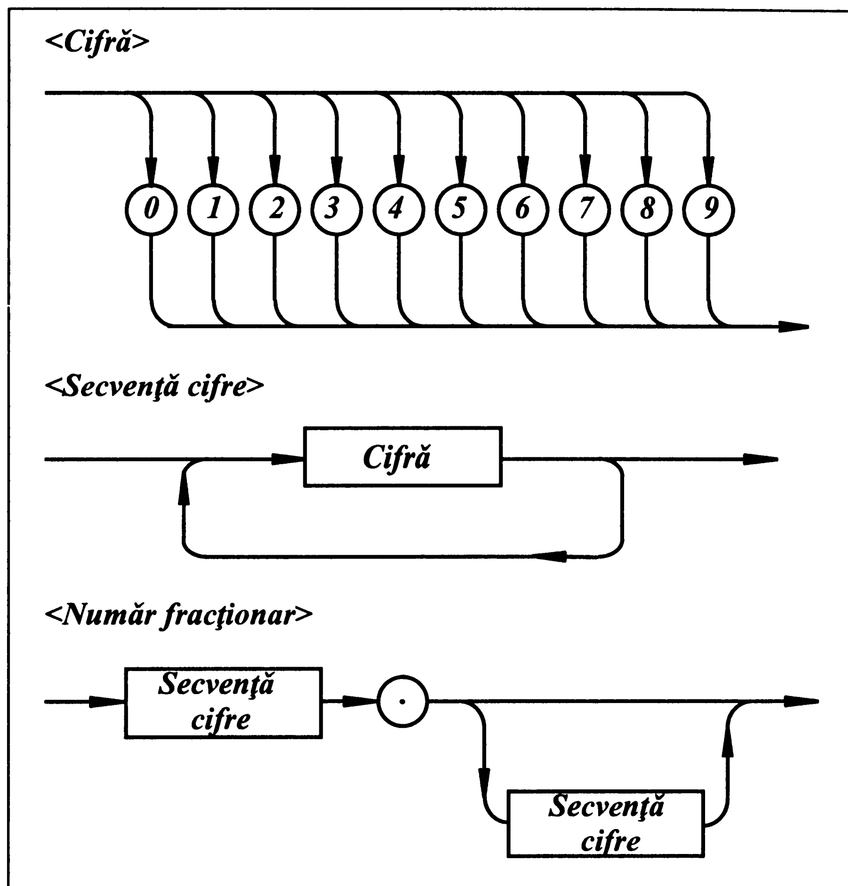


Fig. 1.3. Diagramele sintactice <*Cifră*>, <*Secvență cifre*>, <*Număr fracționar*>

1.4. ALFABETUL LIMBAJULUI

Alfabetul limbajului PASCAL este format din următoarele caractere ale codului *ASCII* (*American Standard Code for Information Interchange*):

- cifrele zecimale;
- literele mari şi mici ale alfabetului englez;
- semnele de punctuaţie;
- operatorii aritmetici şi logici;
- caractere de control şi editare (spaţiu, sfîrşit de linie sau retur de car etc.).

În unele construcţii ale limbajului pot fi folosite şi literele alfabetelor naţionale, de exemplu literele ă, â, î, ş, ţ ale alfabetului român.

1.5. VOCABULARUL LIMBAJULUI

Cele mai simple elemente, alcătuite din caractere şi care au semnificaţie lingvistică, se numesc lexeme sau unităţi lexicale. Acestea formează **vocabularul** limbajului PASCAL.

Distingem următoarele **unităţi lexicale**:

- simboluri speciale şi cuvinte-cheie;
- identificatori;
- numere;
- şiruri de caractere;
- etichete;
- directive.

1.6. SIMBOLURILE SPECIALE ŞI CUVINTELE-CHEIE

Simbolurile speciale sînt formate din unul sau două caractere:

+	plus	<	mai mic
–	minus	>	mai mare
*	asterisc	[	paranteză pătrată din stînga
/	bară	]	paranteză pătrată din dreapta
=	egal	(	paranteză rotundă din stînga
,	virgulă	)	paranteză rotundă din dreapta
:	două puncte	;	punct şi virgulă
.	punct	^	putere
@	la	$	dolar

{	acoladă din stînga	<=	mai mic sau egal
}	acoladă din dreapta	=>	mai mare sau egal
#	număr	:=	atribuire
..	puncte de suspensie	<>	neegal
(*	echivalentul acoladei {	(.	echivalentul parantezei [
*)	echivalentul acoladei }	.)	echivalentul parantezei]

Menţionăm că dacă un simbol special este alcătuit din două caractere, de exemplu <= sau :=, între ele nu trebuie să apară nici un spaţiu intermediar.

Cuvintele-cheie sînt formate din două sau mai multe litere:

and	şi	**nil**	zero
array	tablou	**not**	nu
begin	început	**of**	din
case	caz	**or**	sau
const	constante	**packed**	împachetat
div	cîtul împărţirii	**procedure**	procedură
do	execută	**program**	program
downto	în descreştere la	**record**	articol (înregistrare)
else	altfel	**repeat**	repetare
end	sfîrşit	**set**	mulţime
file	fişier	**then**	atunci
for	pentru	**to**	la
function	funcţie	**type**	tip
goto	treci la	**until**	pînă ce
if	dacă	**var**	variabile
in	în	**while**	cît
label	etichetă	**with**	cu
mod	restul împărţirii		

Cuvintele-cheie sînt rezervate şi nu pot fi folosite în alt scop decît cel dat prin definiţia limbajului.

Unităţile lexicale în studiu se definesc cu ajutorul următoarelor formule BNF:

<*Simbol special*> ::= + | − | * | / | = | < | > |] | [| , | (|) |
: | ; | ^ | . | @ | { | } | $ | # | <= | => | <> | := |
.. | <*Cuvînt-cheie*> | <*Simbol echivalent*>

<*Simbol echivalent*> ::= (* | *) | (. | .)

<*Cuvînt-cheie*> ::= **and** | **array** | **begin** | **case** | **const** | **div** | **do** | **downto** | **else** | **end** | **file** | **for** | **function** | **goto** | **if** | **in** | **label** | **mod** | **nil** | **not** | **of** | **or** | **packed** | **procedure** | **program** | **record** | **repeat** | **set** | **then** | **to** | **type** | **until** | **var** | **while** | **with**

De reţinut că simbolurile { , } , [şi] utilizate în notaţia BNF sînt în acelaşi timp şi elemente ale vocabularului PASCAL. Pentru a evita confuziile, aceste simboluri, ca elemente ale vocabularului, pot fi redate prin simbolurile echivalente `(*`, `*)`, `(.` şi respectiv `.)`.

Întrebări şi exerciţii

❶ Memoraţi cuvintele-cheie ale limbajului PASCAL.

❷ Care este diferenţa dintre caractere şi simboluri?

❸ Desenaţi diagramele sintactice pentru unităţile lexicale *<Simbol special>*, *<Simbol echivalent>* şi *<Cuvînt-cheie>*.

1.7. IDENTIFICATORI

Identificatorii sînt unităţi lexicale care desemnează variabile, constante, funcţii, programe ş. a.

Un identificator începe cu o literă, care poate fi urmată de orice combinaţie de litere şi cifre. Lungimea identificatorilor nu este limitată, dar sînt semnificative doar primele 63 de caractere.

Amintim formulele BNF care definesc unitatea lexicală *<Identificator>*:

<Cifră> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Literă> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z

<Identificator> ::= *<Literă>* { *<Literă>* | *<Cifră>* }

Exemple de identificatori:

```
x   y   z   x1   y10  z01b   lista
listaelevilor       listatelefoanelor
registru            adresa
adresadomiciliu     casa10
```

În construcţiile gramaticale ale limbajului PASCAL, cu excepţia şirurilor de caractere, literele mari şi mici se consideră echivalente. Prin urmare, sînt echivalenţi şi identificatorii

`x` şi `X`, `y` şi `Y`, `z` şi `Z`, `x1` şi `X1`, `y10` şi `Y10`, `z01b`, `Z01b`, `Z01B` şi `z01B`, `lista`, `Lista`, `LIsta`, `ListA`, `LISTa` etc.

Utilizarea literelor mari şi mici ne permite să scriem identificatorii mai citeţ, de exemplu:

```
ListaElevilor ListaTelefoanelor AdresaDomiciliu
```

Menţionăm că în construcţiile de bază ale limbajului PASCAL nu se utilizează literele ă, â, î, ş, ţ ale alfabetului român. Prin urmare, în scrierea identificatorilor semnele diacritice respective vor fi omise.

Exemple:

```
Suprafata        Numar            NumarElevi
Patrat           SirDeCaractere   NumarIncercari
```

Întrebări şi exerciţii

❶ Desenaţi diagramele sintactice pentru unităţile gramaticale <*Cifră*>, <*Literă*> şi <*Identificator*>.

❷ Care din secvenţele ce urmează sînt conforme definiţiei unităţii lexicale <*Identificator*>:

```
x1    Suprafata    A1x       Dreptunghi
X1    SUPRAFATA    ListaA    iI
1x    radacina     Lista1    I1j
1X    radacina     B-1       Luni
xy    R1           abc       Luna
```

Pentru secvenţele corecte indicaţi drumurile respective din diagrama sintactică <*Identificator*>.

❸ Găsiţi perechile de identificatori echivalenţi:

```
x101               CERCURI
ya15               SirDeCaractere
radacinaX1         Triunghiuri
radacinaX2         RegistruClasa10
triunghi           zilE
cerc               X101
sirdecaractere     RegistruClasa10
registruclasa10    radaciniX1X2
COTIDIAN           RADACINAX1
ZILE               yA101
```

❹ Care este destinaţia identificatorilor din programele PASCAL?

❺ Pentru a găsi soluţiile x_1, x_2 ale ecuaţiei pătrate $ax^2 + bx + c = 0$, mai întîi se calculează discriminantul *d*. Propuneţi cîteva variante de reprezentare a coeficienţilor *a*, *b*, *c*, a discriminantului *d* şi soluţiilor x_1, x_2 *prin identificatori.*

1.8. NUMERE

Numerele pot fi întregi sau reale. În mod obişnuit, se foloseşte sistemul zecimal de numeraţie. În *fig. 1.4* sînt prezentate diagramele sintactice pentru unităţile lexicale <*Număr întreg*> şi <*Număr real*>.

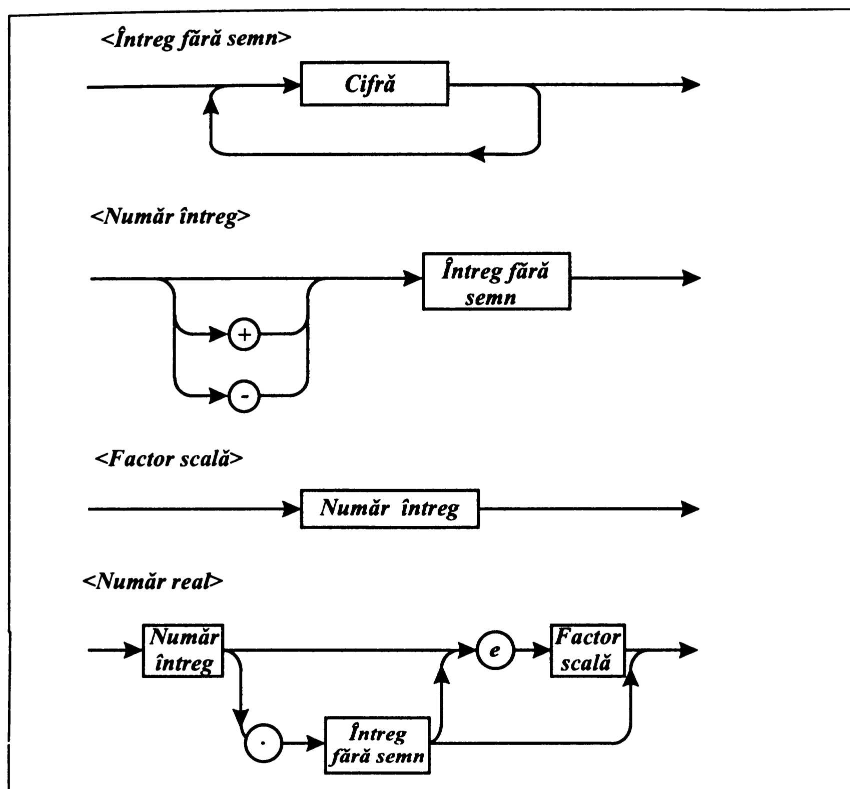


Fig. 1.4. Diagramele sintactice *<Număr întreg>* şi *<Număr real>*

Exemple de numere întregi:

```
23      -23     0023   +023      -0023
318     00318  +0318   -318      -0318
1996   +1996   -1996   0001996   +001996
```

În cazul numerelor reale partea fracţionară se separă de partea întreagă prin punct. Punctul zecimal trebuie să fie precedat şi urmat de cel puţin o cifră zecimală.

Exemple de numere reale:

```
3.1415   +3.04   0.0001
283.19   -3.04   -0.0001
-256.19  28.17   +0.0001
```

În scrierea numerelor reale se poate utiliza şi un **factor de scală**. Acesta este un număr întreg precedat de litera `e` (sau `E`) şi indică că

numărul urmat de factorul de scală se înmulţeşte cu 10 la puterea respectivă.

Exemple:

forma uzuală	*notaţia PASCAL*
$8{,}12 \cdot 10^{-5}$	`8.12e-5`
$749{,}512 \cdot 10^{8}$	`749.512e+8`
$-0{,}0823 \cdot 10^{-12}$	`-0.0823e-12`
$3250{,}4 \cdot 10^{6}$	`3250.4e06`
$3{,}421 \cdot 10^{16}$	`3.421e16`

Evident, `8.12e-05`, `812e-7`, `0.812e-4`, `81.2e-6` reprezintă una şi aceeaşi valoare $8{,}12 \cdot 10^{-5}$.

Întrebări şi exerciţii

❶ Care din secvenţele de caractere ce urmează sînt conforme definiţiei unităţii lexicale <*Număr întreg*>?

```
-418    2469    32,14    -621    +621
0-418   -6210   +00621   32,014  -00418
621+    0+2469  24690    -719    -00621
```

Găsiţi numerele întregi care reprezintă una şi aceeaşi valoare.

❷ Pornind de la diagramele sintactice din *fig. 1.4*, scrieţi formulele BNF pentru definirea unităţii lexicale <*Număr întreg*>.

❸ Care din secvenţele ce urmează sînt conforme definiţiei unităţii lexicale <*Număr real*>?

```
3.14         281.3         0,618284e00
2.514e+5     591328        1961.
591328E+3    2514e+2       28130E-2
.000382      -464.597e+3   591.328
0.1961E+4    +519.328e-4   -658.14e-6
+314629.     591328e-3     2514e+2
0.000314E4   28130e-2      618.248e-3
```

Găsiţi numerele reale care reprezintă una şi aceeaşi valoare. Scrieţi aceste numere în forma uzuală.

❹ Pornind de la diagramele sintactice din *fig. 1.4*, scrieţi formulele BNF pentru definirea unităţii lexicale <*Număr real*>.

❺ Indicaţi pe diagramele sintactice din *fig. 1.4* drumurile care corespund următoarelor numere:

```
-418      281.3      32.014      +0001
1961.0    2.514e+5   591.328     -614.85e-3
2514E+2   -1951.12   +19.511e+2  2013e-4
```

1.9. ŞIRURI DE CARACTERE

Şirurile de caractere sînt şiruri de caractere imprimabile, delimitate de apostrof. În şirul de caractere apostroful apare dublat. Accentuăm că în cazul şirurilor de caractere literele mari şi mici apar drept caractere distincte.

Exemple:

```
'Variabila x'
'Calculul aproximativ'
'APOSTROFUL '' ESTE DUBLAT'
```

Spre deosebire de alte unităţi lexicale ale limbajului PASCAL, în şirurile de caractere pot fi utilizate şi literele ă, â, î, ş, ţ ale alfabetului român. Pentru aceasta e necesar ca pe calculatorul la care lucraţi să fie instalate **programele-pilot** ce asigură introducerea, afişarea şi imprimarea literelor în studiu.

Exemple:

```
'Şir de caractere'
'Limba engleză'
'Suprafaţă'
'Număr încercări'
```

Unitatea lexicală *<Şir de caractere>* se defineşte cu ajutorul următoarelor formule BNF:

<Şir de caractere> ::= ' *<Element şir>* {*<Element şir>*} '

<Element şir> ::= ' ' | *<Orice caracter imprimabil>*

Diagrama sintactică a unităţii lexicale în studiu este prezentată în *fig. 1.5.*

Întrebări şi exerciţii

❶ Indicaţi pe diagramele sintactice din *fig. 1.5* drumurile care corespund următoarelor şiruri de caractere:

```
'variabila z'
''''
'Caracterele ''x'', ''y'''
'UNITĂŢI LEXICALE'
```

❷ Care dintre secvenţele ce urmează sînt conforme definiţiei unităţii lexicale *<Şir de caractere>*:

```
'Număr întreg'              'Anul 1997'
'Sfîrşitul programului'     'Rădăcină pătrată'
'APOSTROF'                    'Anul ‘97'
```

```
''x''                    'Lista telefoanelor'
'funcţie'                '''
```

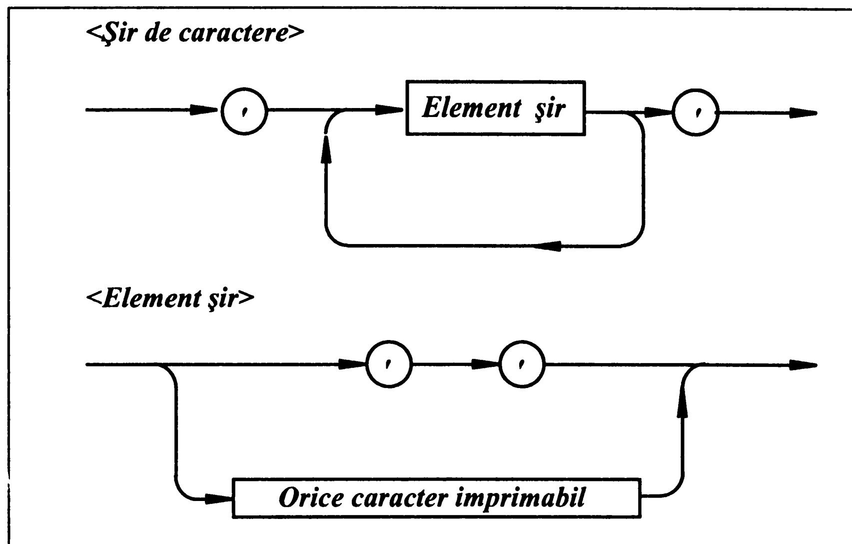


Fig. 1.5. Diagrama sintactică *<Şir de caractere>*

1.10. ETICHETE

Etichetele sînt numere întregi fără semn din domeniul `0, 1, ..., 9999` şi se utilizează pentru a marca instrucţiunile limbajului PASCAL.

Exemple:

```
1  100  999  582  1004
```

Evident, formula BNF care defineşte unitatea lexicală în studiu are forma:

<Etichetă> ::= *<Întreg fără semn>*

1.11. DIRECTIVE

Unitatea lexicală *<Directivă>* se defineşte exact ca identificatorul:

<Directivă> ::= *<Literă>* {*<Literă>* | *<Cifră>*}

Efectiv, directivele sînt cuvinte rezervate care au o semnificaţie specială. Limbajul-standard conţine o singură directivă:

```
forward
```

Aceasta se utilizează la descrierea unor proceduri (subalgoritmi) şi funcţii definite de utilizator.

1.12. SEPARATORI

Orice program PASCAL constă din lexeme şi separatori. Separatorii folosiţi în limbaj sînt spaţiul, sfîrşitul de linie (retur de car) şi comentariul.

La scrierea consecutivă a identificatorilor, a cuvintelor-cheie, a numerelor fără semn şi a directivelor, începutul unei unităţi lexicale ar putea fi interpretat în unele cazuri drept o continuare a celei precedente. Pentru a delimita unităţile lexicale în studiu, între ele se inserează spaţii sau retururi de car.

Exemple:

```
x div y
not x
begin
writeln(x);
writeln(y);
end.
```

Menţionăm că simbolurile speciale compuse din două caractere <=, =>, <>, :=, .. etc., identificatorii, numerele ş.a.m.d. sînt unităţi lexicale ale programului. Prin urmare, nu se pot introduce spaţii sau retururi de car între caracterele componente.

Exemple:

Corect	*Incorect*
CitireDisc	*Citire Disc*
Program	*Pro gram*
:=	*: =*
..	*. .*
345	*3 45*
downto	*down to*
begin	*be gin*

Comentariile sînt secvenţe de caractere precedate de { şi urmate de } .

Exemple:

```
{ Program elaborat de Radu Ion }
{ Introducerea datelor iniţiale }
{ Datele iniţiale se introduc de la tastatură.
Rezultatele vor fi afişate pe ecran şi tipărite la
imprimantă peste 3-4 minute }
```

Comentariile nu influenţează în nici un fel derularea programelor PASCAL şi se utilizează pentru a include în ele unele precizări, explicaţii, informaţii suplimentare etc. Evident, comentariile sînt predestinate nu calculatorului, ci persoanelor care citesc programul respectiv.

Accentuăm că utilizarea raţională a comentariilor, spaţiilor şi retururilor de car asigură scrierea unor programe lizibile (uşor de citit).

Capitolul 2

TIPURI DE DATE SIMPLE

2.1. CONCEPTUL DE DATĂ

Informaţia ce urmează să fie supusă unei prelucrări este accesibilă calculatorului în formă de date. **Datele** sînt constituite din cifre, litere, semne, numere, şiruri de caractere ş.a.m.d.

Într-un limbaj cod-calculator datele sînt reprezentate prin secvenţe de cifre binare. De exemplu, la nivelul procesorului numărul natural 1039 se reprezintă în sistemul de numeraţie binar ca:

`10000001111`

Într-un program PASCAL datele sînt reprezentate prin **mărimi**, şi anume, prin variabile şi constante. Pentru a scuti utilizatorul de toate detaliile legate de reprezentarea internă a datelor, în PASCAL se utilizează diverse tipuri de date.

Prin **tip de date** se înţelege o **mulţime de valori** şi o **mulţime de operaţii** care pot fi efectuate cu valorile respective.

De exemplu, în versiunea *Turbo PASCAL 7.0* tipul `integer` include mulţimea numerelor întregi

`{-32768, -32767, ..., -2, -1, 0, 1, 2, ..., 32767}.`

Cu aceste numere pot fi efectuate următoarele operaţii:

+ adunarea;
- scăderea;
* înmulţirea;
`mod` restul împărţirii;
`div` cîtul împărţirii ş.a.

Tipul de date `real` (real) include o submulţime a numerelor reale, operaţiile +, -, *, / (împărţirea) ş.a.

Operaţiile `mod` şi `div`, admise în tipul de date `integer`, sînt inadmisibile în tipul de date `real`.

Conceptul de dată realizat în limbajul PASCAL presupune:

1) fiecare mărime (variabilă sau constantă) într-un program în mod obligatoriu se asociază cu un anumit tip de date;

2) tipul unei variabile defineşte mulţimea de valori pe care le poate lua variabila şi operaţiile care pot fi efectuate cu aceste valori;

3) există tipuri de date de interes general, definiţia cărora se consideră cunoscută: `integer`, `real`, `char` (caracter), `boolean` (logic), `text` ş.a.;

4) pe baza tipurilor cunoscute programatorul poate crea tipuri noi, adecvate informaţiilor de prelucrat.

Tipul variabilelor se declară explicit cu ajutorul cuvîntului-cheie **var**.

Exemplu:

```
Var x, y : integer;
    z : real;
```

Tipul unei **constante** se declară implicit prin forma ei textuală.

De exemplu, `10` este un număr de tip `integer`, iar `10.0` este un număr de tip `real`.

Întrebări şi exerciţii

❶ Cum se reprezintă datele în limbajul cod-calculator? Care sînt avantajele şi deficienţele acestei reprezentări?

❷ Cum se reprezintă datele într-un program PASCAL? Care este diferenţa dintre variabile şi constante?

❸ Explicaţi semnificaţia termenului *tip de date*. Daţi exemple.

❹ Cum se asociază o variabilă la un anumit tip de date?

❺ Determinaţi tipul variabilelor *r, s, t, x, y* şi *z* din declaraţia ce urmează:

```
var  r, y : integer;
     s, z : real;
     t, x : boolean;
```

❻ Scrieţi o declaraţie care ar defini *a, b* şi *c* ca variabile întregi, iar *p* şi *q* ca variabile *text*.

❼ Precizaţi tipul următoarelor numere:

–301; –301.0; +6100; –61.00e+2; 3.14.

2.2. TIPUL DE DATE integer

Mulţimea de valori ale tipului de date `integer` este formată din numerele întregi care pot fi reprezentate pe calculatorul-gazdă al limbajului. Valoarea maximă poate fi referită prin constanta `MaxInt`, cunoscută oricărui program PASCAL. De obicei, valoarea minimă, admisă de tipul de date în studiu, este `-MaxInt` sau `-(MaxInt+1)`.

Programul ce urmează afişează pe ecran valoarea constantei predefinite MaxInt.

```
Program P2;
  {Afişarea constantei predefinite MaxInt }
  begin
    writeln('MaxInt=', MaxInt);
  end.
```

Pe un calculator IBM PC, versiunea Turbo PASCAL 7.0, constanta MaxInt are valoarea 32767, iar mulţimea de valori ale tipului integer este:

{-32768, -32767, ..., -2, -1, 0, 1, 2, ..., 32767}.

Operaţiile care se pot face cu valorile întregi sînt: +, -, *, mod, div ş.a. Rezultatele acestor operaţii pot fi vizualizate cu ajutorul programului P3.

```
Program P3;
  { Operaţii cu date de tipul integer }
  var x, y, z : integer;
  begin
    writeln('Introduceţi numerele întregi x, y:');
    readln(x,y);
    writeln('x=', x);
    writeln('y=', y);
    z:=x+y; writeln('x+y=', z);
    z:=x-y; writeln('x-y=', z);
    z:=x*y; writeln('x*y=', z);
    z:=x mod y; writeln('x mod y=', z);
    z:=x div y; writeln('x div y=', z);
  end.
```

Evident, rezultatele operaţiilor +, -, * cu valori întregi trebuie să aparţină mulţimii de valori ale tipului de date integer. Dacă programatorul nu acordă atenţia cuvenită acestei reguli, apar erori de depăşire. Aceste erori vor fi semnalate în procesul compilării sau execuţiei programului respectiv. Pentru exemplificare prezentăm programele P4 şi P5:

```
Program P4;
  {Eroare de depăşire semnalată în procesul
  compilării}
  var x : integer;
  begin
    x:=MaxInt+1; {Eroare, x>MaxInt }
    writeln(x);
  end.
```

```
Program P5;
 { Eroare de depăşire semnalată în procesul execuţiei}
var x, y : integer;
begin
 x:=MaxInt;
 y:=x+1; { Eroare, y>MaxInt }
 writeln(y);
end.
```

Întrebări şi exerciţii

❶ Care este mulţimea de valori ale tipului de date `integer`? Ce operaţii se pot face cu aceste valori?
❷ Cînd apar erori de depăşire? Cum se depistează aceste erori?
❸ Determinaţi valoarea constantei `MaxInt` a versiunii PASCAL cu care lucraţi D-voastră.
❹ Se consideră programele :

```
Program P6;
  { Eroare de depăşire }
  var x : integer;
  begin
    x:=-2*MaxInt;
    writeln(x);
  end.
```

```
Program P7;
    { Eroare de depăşire }
  var x, y : integer;
  begin
    x:=-MaxInt;
    y:=x-10;
    writeln(y);
  end.
```

Cînd vor fi semnalate erori de depăşire: la compilare sau la execuţie?
❺ Daţi exemple de valori ale variabilelor `x` şi `y` din programul `P3` pentru care apar erori de depăşire.

2.3. TIPUL DE DATE `real`

Mulţimea de valori ale tipului de date în studiu este formată din numerele reale care pot fi reprezentate pe calculatorul-gazdă al limbajului.

De exemplu, în versiunea Turbo PASCAL 7.0 domeniul de valori al tipului real este $2{,}9 \cdot 10^{-39}$, ..., $1{,}7 \cdot 10^{38}$, numerele fiind reprezentate cu o precizie de 11–12 cifre zecimale.

În programul ce urmează variabilelor reale x, y şi z li se atribuie valorile, respectiv, 1,1, $-6{,}14 \cdot 10^{8}$ şi $90{,}3 \cdot 10^{-29}$, ulterior afişate pe ecran.

```
Program P8;
  { Date de tip real }
  var x, y, z : real;
  begin
    x:=1.1;
    y:=-6.14e8;
    z:=90.3e-29;
    writeln('x=',x);
    writeln('y=',y);
    writeln('z=',z);
  end.
```

Amintim că la scrierea numerelor reale virgula zecimală este redată prin punct, iar puterea lui 10 – prin factorul de scală (vezi paragraful 1.8).

Operaţiile care se pot face cu valorile reale sînt +, –, *, / (împărţirea) ş.a.

Operaţiile asupra valorilor reale sînt în general aproximative datorită erorilor de rotunjire. Evident, rezultatele operaţiilor în studiu trebuie să aparţină domeniului de valori ale tipului de date `real`. În caz contrar, apar erori de depăşire.

Proprietăţile operaţiilor +, –, * şi / pot fi studiate cu ajutorul programului ce urmează:

```
Program P9;
  {Operaţii cu date de tipul real }
  var x, y, z : real;
  begin
    writeln('Introduceţi numerele reale x, y:');
    readln(x,y);
    writeln('x=', x);
    writeln('y=', y);
    z:=x+y; writeln('x+y=', z);
    z:=x-y; writeln('x-y=', z);
    z:=x*y; writeln('x*y=', z);
    z:=x/y; writeln('x/y=', z);
  end.
```

În *tabelul 2.1* sînt prezentate datele afişate de programul P9 (versiunea Turbo PASCAL 7.0) pentru unele valori ale variabilelor x şi y. Se observă că rezultatele operaţiilor x+y şi x-y din primele două linii ale *tabelului 2.1* sînt exacte. În cazul valorilor x = 1,0 , y = $1{,}0 \cdot 10^{-11}$ (linia 3 a tabelului în studiu) rezultatul adunării este aproxi-

mativ, iar cel al scăderii – exact. Ambele rezultate din linia a patra sînt aproximative. În cazul valorilor x = y = $1{,}7 \cdot 10^{38}$ (linia 5) are loc o depăşire la efectuarea adunării. Pentru valorile x = $3{,}1 \cdot 10^{-39}$, y = $3{,}0 \cdot 10^{-39}$ (linia 6) rezultatul adunării este exact, iar rezultatul scăderii este aproximativ.

Tabelul 2.1. Rezultatele programului P9

Nr. crt.	*x*	*y*	*x+y*	*x-y*
1	*1,0*	*1,0*	**2.0000000000 E+00**	**0.0000000000 E+00**
2	*1,0*	$1 \cdot 10^{-10}$	**1.0000000001 E+00**	**9.9999999990 E-01**
3	*1,0*	$1{,}0 \cdot 10^{-11}$	**1.0000000000 E+00**	**9.9999999999 E-01**
4	*1,0*	$1{,}0 \cdot 10^{-12}$	**1.0000000000 E+00**	**1.0000000000 E+00**
5	$1{,}7 \cdot 10^{38}$	$1{,}7 \cdot 10^{38}$	***depăşire***	**0.0000000000 E+00**
6	$3{,}1 \cdot 10^{-39}$	$3{,}0 \cdot 10^{-39}$	**6.1000000000 E-39**	**0.0000000000 E+00**

Însumîndu-se, erorile de calcul, proprii tipului de date `real`, pot compromite rezultatele execuţiei unui program. Evaluarea şi, dacă e necesar, suprimarea erorilor semnificative cade în sarcina programatorului.

Întrebări şi exerciţii

❶ Cum se scriu numerele reale în limbajul PASCAL?

❷ Determinaţi domeniul de valori ale tipului de date `real` din versiunea PASCAL cu care lucraţi. Care este precizia numerelor respective?

❸ Ce operaţii se pot face cu datele de tip `real`? Sînt oare exacte aceste operaţii?

❹ Lansaţi în execuţie programul P9 pentru următoarele valori ale variabilelor x, y:

a) x = 2,0; y = –3,0;
b) x = $14{,}3 \cdot 10^{2}$; y = $15{,}3 \cdot 10^{-3}$;
c) x = 3,0; y = $2{,}0 \cdot 10^{12}$;
d) x = 3,0; y = $2{,}0 \cdot 10^{-12}$;
e) x = $2{,}9 \cdot 10^{-39}$; y = $6{,}4 \cdot 10^{-3}$;
f) x = $7{,}51 \cdot 10^{21}$; y = $-8{,}64 \cdot 10^{17}$;
g) x = 1,0; y = $2{,}9 \cdot 10^{-39}$.

Verificaţi rezultatele operaţiilor respective. Explicaţi mesajele afişate pe ecran.

❺ Care sînt cauzele erorilor de calcul cu date de tip `real`?

2.4. TIPUL DE DATE boolean

Tipul de date `boolean` (logic) include valorile de adevăr `false` (fals) şi `true` (adevărat). În programul ce urmează variabilei `x` i se atribuie consecutiv valorile `false` şi `true`, afişate ulterior pe ecran.

```
Program P10;
  { Date de tip boolean }
  var x : boolean;
  begin
    x:=false;
    writeln(x);
    x:=true;
    writeln(x);
  end.
```

Operaţiile predefinite ale tipului de date `boolean` sînt:

not negaţia (inversia logică, operaţia logică *NU*);
and conjuncţia (produsul logic, operaţia logică *ŞI*);
or disjuncţia (suma logică, operaţia logică *SAU*).

Tabelele de adevăr ale operaţiilor în studiu sînt prezentate în *fig. 2.1*.

x	**not** *x*
false	*true*
true	*false*

x	*y*	*x* **and** *y*
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

x	*y*	*x* **or** *y*
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*true*

Fig. 2.1. Tabelele de adevăr ale operaţiilor logice **not**, **and** şi **or**

Proprietățile operațiilor logice not, and și or pot fi studiate cu ajutorul programului P11.

```
Program P11;
  {Operații cu date de tip boolean }
  var x, y, z : boolean;
  begin
    x:=false; y:=false;
    writeln('x=', x, 'y=', y);
    z:=not x; writeln('not x = ', z);
    z:=x and y; writeln('x and y = ', z);
    z:=x or y; writeln('x or y = ', z);
    writeln;
    x:=false; y:=true;
    writeln('x=', x, 'y=', y);
    z:=not x; writeln('not x = ', z);
    z:=x and y; writeln('x and y = ', z);
    z:=x or y; writeln('x or y = ', z);
    writeln;
    x:=true; y:=false;
    writeln('x=', x, 'y=', y);
    z:=not x; writeln('not x = ', z);
    z:=x and y; writeln('x and y = ', z);
    z:=x or y; writeln('x or y = ', z);
    writeln;
    x:=true; y:=true;
    writeln('x=', x, 'y=', y);
    z:=not x; writeln('not x = ', z);
    z:=x and y; writeln('x and y = ', z);
    z:=x or y; writeln('x or y = ', z);
    writeln;
  end.
```

Spre deosebire de variabilele de tip întreg sau real, valorile curente ale variabilelor booleene nu pot fi citite de la tastatură cu ajutorul procedurii-standard readln. Din acest motiv, în programul P11 valorile curente ale variabilelor x și y sînt date prin atribuire.

Întrebări și exerciții

❶ Numiți mulțimea de valori și operațiile cu date de tip boolean.
❷ Memorizați tabelele de adevăr ale operațiilor logice.
❸ Elaborați un program care afișează pe ecran tabelul de adevăr al operației logice not.
❹ Elaborați un program care calculează valorile funcției logice $z = x \& y$ pentru toate valorile posibile ale argumentelor x, y.
❺ Elaborați un program care afișează valorile funcției logice $z = x \vee y$.

2.5. TIPUL DE DATE char

Mulţimea valorilor acestui tip de date este o mulţime finită şi ordonată de caractere. Valorile în studiu se desemnează prin includerea fiecărui caracter între două semne ' (apostrof), de exemplu, *'A'*, *'B'*,*'C'* etc. Însuşi apostroful se dublează, reprezentîndu-se prin ''''.

În programul ce urmează variabilei x de tip char i se atribuie consecutiv valorile *'A'*, *'+'* şi *''''*, afişate ulterior pe ecran.

```
Program P12;
  { Date de tip char }
  var x : char;
  begin
    x:='A';
    writeln(x);
    x:='+';
    writeln(x);
    x:='''';
    writeln(x);
  end.
```

Valorile curente ale unei variabile de tip char pot fi citite de la tastatură cu ajutorul procedurii-standard readln. Pentru exemplificare prezentăm programul P13 care citeşte de la tastatură şi afişează pe ecran valori de tipul char.

```
Program P13;
  {Citirea şi afişarea caracterelor }
  var x : char;
  begin
    readln(x); writeln(x);
    readln(x); writeln(x);
    readln(x); writeln(x);
  end.
```

Caracterele respective se introduc de la tastatură şi se afişează pe ecran fără apostrofurile care le încadrează în textul unui program PASCAL.

De regulă, caracterele unei versiuni concrete a limbajului PASCAL sînt **ordonate** conform tabelului de cod *ASCII* (vezi paragraful 1.4).

Numărul de ordine al oricărui caracter din mulţimea de valori ale tipului char poate fi aflat cu ajutorul funcţiei predefinite ord. De exemplu:

ord(*'A'*)=65,
ord(*'B'*)=66,

ord('C')=67
ş.a.m.d.

Programul ce urmează afişează pe ecran numărul de ordine al patru caractere citite de la tastatură.

```
Program P14;
  { Studierea funcţiei ord }
  var x : char; { caracter }
    i : integer; { număr de ordine }
  begin
    readln(x); i:=ord(x); writeln(i);
    readln(x); i:=ord(x); writeln(i);
    readln(x); i:=ord(x); writeln(i);
    readln(x); i:=ord(x); writeln(i);
  end.
```

Funcţia predefinită chr returnează caracterul care corespunde numărului de ordine indicat. Evident,

chr(65)='A',
chr(66)='B',
chr(67)='C'
ş.a.m.d.

Programul P15 afişează pe ecran caracterele ce corespund numerelor de ordine citite de la tastatură.

```
Program P15;
  { Studierea funcţiei chr }
  var i : integer; { număr de ordine }
    x : char; { caracter }
  begin
    readln(i); x:=chr(i); writeln(x);
    readln(i); x:=chr(i); writeln(x);
    readln(i); x:=chr(i); writeln(x);
    readln(i); x:=chr(i); writeln(x);
  end.
```

Amintim că un set extins *ASCII* include 256 de caractere, numerotate cu 0, 1, 2, ..., 255.

Tipul de date char se utilizează pentru formarea unor structuri de date mai complexe, în particular, a şirurilor de caractere.

Întrebări şi exerciţii

❶ Care este mulţimea de valori ale tipului de date char?
❷ Cum este ordonată mulţimea de valori ale tipului char?
❸ Determinaţi numerele de ordine ale următoarelor caractere:

- cifrele zecimale;
- literele mari ale alfabetului englez;
- semnele de punctuaţie;
- operatorii aritmetici şi logici;
- caracterele de control şi editare;
- literele alfabetului român (dacă sînt implementate pe calculatorul D-voastră).

❹ Determinaţi caracterele care corespund următoarelor numere de ordine:

`77, 109, 79, 111, 42, 56, 91, 123.`

❺ Elaboraţi un program care afişează pe ecran setul de caractere al calculatorului cu care lucraţi.

2.6. TIPURI DE DATE *ENUMERARE*

Tipurile `integer`, `real`, `boolean` şi `char`, studiate pînă acum, sînt tipuri predefinite, cunoscute oricărui program PASCAL. În completare la tipurile predefinite, programatorul poate defini şi utiliza tipuri proprii de date, în particular, tipuri *enumerare.*

Un tip ***enumerare*** include o mulţime ordonată de valori specificate prin identificatori. Denumirea unui tip de date *enumerare* şi mulţimea lui de valori se indică în partea declarativă a programului după cuvîntul-cheie **type** (tip).

Exemplu:

```
type
  Culoare   = (Galben, Verde, Albastru, Violet);
  Studii    = (Elementare, Medii, Superioare);
  Raspuns   = (Nu,Da);
```

Primul identificator din lista de enumerare desemnează cea mai mică valoare, cu numărul de ordine zero. Identificatorul al doilea va avea numărul de ordine unu, al treilea – numărul doi ş.a.m.d. Numărul de ordine al unei valori poate fi aflat cu ajutorul funcţiei predefinite `ord`.

Exemple:

```
ord(Galben)=0,
ord(Verde)=1,
ord(Albastru)=2,
ord(Violet)=3,
ord(Elementare)=0,
ord(Medii)=1
ş.a.m.d.
```

Programul ce urmează afişează pe ecran numerele de ordine ale valorilor tipului de date Studii.

```
Program P16;
  { Tipul de date Studii }
  type Studii=(Elementare, Medii, Superioare);
  var i : integer; { număr de ordine }
  begin
    i:=ord(Elementare); writeln(i);
    i:=ord(Medii); writeln(i);
    i:=ord(Superioare);
    writeln(i);
  end.
```

Variabilele de tip *enumerare* se declară cu ajutorul cuvîntului-cheie **var**. Ele pot lua numai valori din lista de enumerare a tipului de date cu care sînt asociate.

În programul ce urmează variabila x ia valoarea Albastru; variabila y ia valoarea Nu. Numerele de ordine ale acestor valori se afişează pe ecran.

```
Program P17;
  { Variabile de tip enumerare }
  type  Culoare=(Galben, Verde, Albastru, Violet);
        Raspuns=(Nu, Da);
  var x : Culoare; { variabilă de tip Culoare }
      y : Raspuns; { variabilă de tip Raspuns }
      i : integer; { număr de ordine }
  begin
    x:=Albastru;
    i:=ord(x); writeln(i);
    y:=Nu; i:=ord(y); writeln(i);
  end.
```

În cazurile în care într-un program PASCAL se definesc mai multe tipuri de date, listele de enumerare nu trebuie să conţină identificatori comuni.

De exemplu, declaraţia

```
type Studii=(Elementare, Medii, Superioare);
     Grade=(Inferioare, Superioare)
```

este incorectă, întrucît identificatorul Superioare apare în ambele liste.

Valorile curente ale variabilelor de tip *enumerare* nu pot fi citite de la tastatură sau afişate pe ecran cu ajutorul procedurilor-standard readln şi writeln. Totuşi utilizarea tipurilor de date în studiu permite elaborarea unor programe lizibile, simple şi eficiente.

Întrebări şi exerciţii

❶ Cum se defineşte un tip de date *enumerare*? Care este mulţimea de valori ale unui tip *enumerare*?

❷ Contează oare ordinea în care apar identificatorii într-o listă de enumerare?

❸ Elaboraţi un program care afişează pe ecran numerele de ordine ale valorilor următoarelor tipuri de date:

```
Continente= (Europa,Asia,Africa,AmericaDeNord,
             AmericaDeSud,Australia,Antarctida);
Sex=(Barbat,Femeie);
PuncteCardinale=(Nord,Sud,Est,Vest);
Etaje=(Unu,Doi,Trei,Patru,Cinci).
```

❹ Numiţi tipul fiecărei variabile din programul P18:

```
Program P18;
  type Litere=(A, B, C, D, E, F, G);
  var x : Litere; y : char; i : integer;
  begin
    x:=A; i:=ord(x); writeln(i);
    y:='A'; i:=ord(y); writeln(i);
  end.
```

Ce va afişa pe ecran acest program?

❺ Se consideră declaraţiile:

```
type Culoare=(Galben, Verde, Albastru, Violet);
     Fundal=(Alb, Negru, Gri);
var  x, y : Culoare;
     z : Fundal;
```

Care din instrucţiunile ce urmează sînt corecte?

a) `x:=Verde;` e) `y:=Gri;`
b) `y:=Negru;` f) `z:=Violet;`
c) `z:=Alb;` g) `x:=Albastru;`
d) `x:=Gri;` h) `y:=Azuriu;`

❻ Inseraţi înainte de cuvîntul-cheie `end` al programului `P17` una din liniile ce urmează:

```
readln(x);
writeln(x);
```

Explicaţi mesajele afişate pe ecran în procesul compilării programului modificat.

2.7. TIPURI DE DATE *SUBDOMENIU*

Un tip de date *subdomeniu* include o submulţime de valori ale unui tip deja definit, denumit tip de bază. Tipul de bază trebuie să fie `integer`, `boolean`, `char` sau *enumerare*.

Denumirea unui tip de date *subdomeniu*, valoarea cea mai mică şi valoarea cea mai mare (în sensul numărului de ordine) se indică în partea declarativă a programului după cuvîntul-cheie **type**.

Exemple:

```
1)  type Indice=1..10;
         Litera='A'..'Z';
         Cifra='0'..'9';
```

Tipul `Indice` este un subdomeniu al tipului predefinit `integer`. Tipurile `Litera` şi `Cifra` sînt subdomenii ale tipului predefinit `char`.

```
2)  type Zi=(L, Ma, Mi, J, V, S, D);
         ZiDeLucru=L..V;
         ZiDeOdihna=S..D;
```

Tipurile `ZiDeLucru` şi `ZiDeOdihna` sînt subdomenii ale tipului enumerare `Zi`, definit de utilizator.

```
3)  type T1=(A, B, C, D, E, F, G, H);
         T2=A..F;
         T3=C..H;
```

Tipurile `T2` şi `T3` sînt subdomenii ale tipului *enumerare* `T1`.

Tipurile de bază ale tipurilor de date *subdomeniu* din exemplele în studiu sînt:

tip enumerare	*tipul de bază*
Indice	integer
Litera	char
Cifra	char
ZiDeLucru	Zi
ZiDeOdihna	Zi
T2	T1
T3	T1

Variabilele unui tip de date *subdomeniu* se declară cu ajutorul cuvîntului-cheie `var`. O variabilă de tip *subdomeniu* moşteneşte toate proprietăţile variabilelor tipului de bază, dar valorile ei trebuie să fie numai din intervalul specificat. În caz contrar este semnalată o eroare şi programul se opreşte.

Exemplu:

```
Program P19;
  { Valorile variabilelor de tip subdomeniu }
  type Indice=1..10;
       Zi=(L, Ma, Mi, J, V, S, D);
       ZiDeLucru=L..V;
       ZiDeOdihna=S..D;
  var i  : Indice; { valori posibile: 1,2, ...,10 }
      z  : Zi;     { valori posibile: L,Ma, ...,D }
      zl : ZiDeLucru;{ valori posibile: L,Ma, ...,V }
      zo : ZiDeOdihna;{ valori posibile: S, D }
begin
    i:=5; i:=11; { Eroare, i>10 }
    z:=L; zl:=J; zl:=S; { Eroare, zl>V }
    zo:=S; zo:=V; { Eroare, zo<S }
    writeln('Sfîrşit');
  end.
```

Programul `P20` demonstrează cum tipul `Pozitiv` moşteneşte proprietăţile tipului de bază `integer`.

```
Program P20;
  { Tipul Pozitiv moşteneşte proprietăţile tipului
  integer }
  type Pozitiv=1..32767;
  var x, y, z : Pozitiv;
  begin
    writeln('Introduceţi numerele pozitive x, y:');
    readln(x,y);
    writeln('x=', x);
    writeln('y=', y);
    z:=x+y; writeln('x+y=', z);
    z:=x-y; writeln('x-y=', z);
    z:=x*y; writeln('x*y=', z);
    z:=x mod y; writeln('x mod y=', z);
    z:=x div y; writeln('x div y=', z);
  end.
```

Se observă că operaţiile `+`, `-`, `*`, `mod` şi `div` ale tipului de bază `integer` sînt moştenite de tipul *enumerare* `Pozitiv`. Dar, spre deosebire de variabilele de tip `integer`, variabilele de tip `Pozitiv` nu pot lua valori negative.

Utilizarea tipurilor de date *subdomeniu* face programele mai intuitive şi simplifică verificarea lor.

Subliniem faptul că în limbajul PASCAL nu este permisă definirea unui subdomeniu al tipului `real`, deoarece valorile acestuia nu au numere de ordine.

Întrebări şi exerciţii

❶ Cum se defineşte un tip *subdomeniu*? Care este mulţimea de valori ale unui tip *subdomeniu*?

❷ Numiţi tipul de bază al fiecărui tip *subdomeniu*:

```
type T1=(A,B,C,D,E,F,G,H);
     T2=-60..60;
     T3=5..9;
     T4='5'..'9';
     T5=A..E;
     T6='A'..'E';
```

❸ Ce valori poate lua fiecare variabilă din următoarele declaraţii:

```
type T1=(A,B,C,D,E,F,G,H);
     T2=1..9;
     T3=6..15;
     T4=-100..100;
     T5='A'..'Z';
     T6='0'..'9';
     T7=C..F;
var  i:integer;
     j:T2;
     k:T3;
     m:T4;
     q:char;
     p:T5;
     r:T6;
     s:T1;
     t:T7;
```

Numiţi tipul de bază al fiecărui tip *subdomeniu*. Indicaţi setul de operaţii moştenit de la tipul de bază.

❹ Care din următoarele definiţii sînt corecte? Argumentaţi răspunsul.

```
a) type Lungime=1.0e-2..1.0;
        Latime=1.0e-2..0.5;
b) type Indice=1..10;
        Abatere=+5..-5;
        Deviere=-10..+10;
c) type T1=(A,B,C,D,E,F,G,H);
        T2=C..H;
        T3=F..B;
d) type Luni=   (Ianuarie,Februarie,Martie,Aprilie,
               Mai,Iunie,Iulie,August,Septembrie,
               Octombrie,Noiembrie,Decembrie);
        LuniDeIarna=(Decembrie..Februarie);
        LuniDePrimavara=(Martie..Mai);
        LuniDeVara=(Iunie..August);
        LuniDeToamna=(Septembrie..Noiembrie);
```

❺ Se consideră următorul program:

```
Program P21;
  type Indice=1..10;
  var i, j, k, m : Indice;
  begin
    writeln('Introduceţi indicii i, j:');
    readln(i, j);
    k:=i+j; writeln('k=', k);
    m:=i-j; writeln('m=', m);
  end.
```

Pentru care valori ale variabilelor i, j se vor declanşa erori de execuţie?

a) i=3, j=2;
b) i=7, j=4;
c) i=4, j=7;
d) i=6, j=3;
e) i=2, j=2;
f) i=3, j=11;
g) i=8, j=4;
h) i=5, j=3.

❻ Se consideră programul P20. După lansarea în execuţie utilizatorul introduce x=1, y=2. Evident, x-y= –1. Întrucît valoarea –1 nu aparţine tipului de date Pozitiv, la execuţia instrucţiunii

z:=x-y

va surveni o eroare.
Indicaţi instrucţiunile la execuţia cărora se vor declanşa erori, dacă:

a) x=1000, y=1000;
b) x=1000, y=1001;
c) x=1001, y=1000;
d) x=30000, y=30000;
e) x=1, y=2;
f) x=1000, y=100;
g) x=0, y=1;
h) x=1, y=0.

2.8. GENERALITĂŢI DESPRE TIPURILE ORDINALE DE DATE

Tipurile de date integer, boolean, char, *enumerare* şi *subdomeniu* se numesc **tipuri ordinale**. Fiecare valoare a unui tip ordinal are un număr de ordine, definit după cum urmează:

1) numărul de ordine al unui număr de tip integer este însuşi numărul considerat;

2) valorile de adevăr false şi true ale tipului boolean au numerele de ordine, respectiv, 0 şi 1;

3) numărul de ordine al unui caracter (tipul char) este dat de poziţia lui în tabelul de codificare, obişnuit *ASCII*;

4) numărul de ordine al unei valori de tip *enumerare* este dat de poziţia ei în lista de enumerare. De remarcat că valorile unei liste sînt numerotate prin 0, 1, 2, ... ş.a.m.d.;

5) valorile unui tip *subdomeniu* moştenesc numerele de ordine de la tipul de bază.

Numărul de ordine al unei valori de tip ordinal poate fi aflat cu ajutorul funcţiei predefinite `ord`.

Programul `P22` afişează pe ecran numerele de ordine ale valorilor `-32`, `true`, `'A'`, `A` şi `B`.

```
Program P22;
  { Numerele de ordine ale valorilor de tip ordinal }
  type  T1=(A, B, C, D, E, F, G, H);
        T2=B..G;
  begin
    writeln(ord(-32));  {  -32 }
    writeln(ord(true)); {    1 }
    writeln(ord('A'));  {   65 }
    writeln(ord(A));    {    0 }
    writeln(ord(B));    {    1 }
  end.
```

Asupra valorilor oricărui tip ordinal de date sînt permise operaţiile relaţionale cunoscute:

<	*mai mic;*
<=	*mai mic sau egal;*
=	*egal;*
>=	*mai mare sau egal;*
>	*mai mare;*
<>	*diferit.*

Rezultatul unei operaţii relaţionale este de tip `boolean`, `false` sau `true`, în funcţie de numerele de ordine ale valorilor operanzilor.

De exemplu, în prezenţa declaraţiilor

```
type Culoare=(Galben, Verde, Albastru, Violet);
```

rezultatul operaţiei

```
Verde<Violet
```

este `true`, deoarece `ord(Verde)`=1, `ord(Violet)`=3 şi 1 este mai mic ca 3.

Rezultatul operaţiei

```
Galben>Violet
```

este `false`, deoarece `ord(Galben)`=0, `ord(Violet)`=3 şi 0 nu este mai mare ca 3.

Programul ce urmează afişează pe ecran rezultatele operaţiilor relaţionale pentru valorile `Verde` şi `Violet` ale tipului de date `Culoare`.

```
Program P23;
  {Operaţii relaţionale asupra valorilor
```

```
  de tip ordinal }
type Culoare=(Galben, Verde, Albastru, Violet);
begin
  writeln(Verde<Violet);  { true }
  writeln(Verde<=Violet); { true }
  writeln(Verde=Violet);  { false }
  writeln(Verde>=Violet); { false }
  writeln(Verde>Violet);  { false }
  writeln(Verde<>Violet); { true }
end.
```

Pentru tipurile ordinale de date există funcţiile predefinite `pred` (*predecesor*) şi `succ` (*succesor*).

Predecesorul valorii ordinale cu numărul de ordine *i* este valoarea cu numărul de ordine $i - 1$. *Succesorul* valorii ordinale în studiu este valoarea cu numărul de ordine $i + 1$.

De exemplu, pentru valorile tipului ordinal de date `Culoare` obţinem:

```
pred(Verde)= Galben;
succ(Verde)= Albastru;
pred(Albastru)= Verde;
succ(Albastru)= Violet.
```

Evident, valoarea cea mai mică nu are predecesor, iar valoarea cea mai mare nu are succesor.

Programul `P24` afişează pe ecran predecesorii şi succesorii valorilor ordinale `'B'`, 0 şi `'0'`.

```
Program P24;
 { Predecesorii şi succesorii valorilor ordinale }
 begin
   writeln(pred('B')); { 'A' }
   writeln(succ('B')); { 'C' }
   writeln(pred(0));   { -1 }
   writeln(succ(0));   { 1 }
   writeln(pred('0')); { '/' }
   writeln(succ('0')); { '1' }
 end.
```

Se observă că predecesorii şi succesorii valorilor ordinale `0` (tip `integer`) şi `'0'` (tip `char`) nu coincid, deoarece tipurile valorilor respective diferă.

Subliniem faptul că tipul de date `real` nu este un tip ordinal. Prin urmare, asupra valorilor reale nu pot fi aplicate funcţiile `ord` (numărul de ordine), `pred` (predecesor) şi `succ` (succesor). Nerespectarea acestei reguli va declanşa erori.

Întrebări şi exerciţii

❶ Numiţi tipurile ordinale de date. Care sînt proprietăţile lor comune?
❷ Cum se definesc numerele de ordine ale valorilor unui tip ordinal de date?
❸ Ce va afişa pe ecran programul ce urmează?

```
Program P25;
  type Zi=(L, Ma, Mi, J, V, S, D);
  var z1, z2 : Zi;
  begin
    z1:=Ma;
    writeln(ord(z1));
    z2:=pred(z1);
    writeln(ord(z2));
    z2:=succ(z1);
    writeln(ord(z2));
    z1:=Mi; z2:=V;
    writeln(z1<z2);
    writeln(z1>z2);
    writeln(z1<>z2);
  end.
```

❹ Excludeţi din programul de mai jos linia care conţine o eroare:

```
Program P26;
  { Eroare }
  var i : integer;
  begin
    i:=MaxInt;
    writeln(pred(i));
    writeln(succ(i));
  end.
```

Ce rezultate vor fi afişate pe ecran după execuţia programului modificat?
❺ Comentaţi programul ce urmează:

```
Program P27;
  { Eroare }
  var i : integer; x : real;
  begin
    i:=1; x:=1.0;
    writeln(ord(i));
    writeln(ord(x));
    writeln(pred(i));
    writeln(pred(x));
    writeln(succ(i));
    writeln(succ(x));
  end.
```

Excludeţi liniile care conţin erori. Ce rezultate vor fi afişate pe ecran după execuţia programului modificat?

2.9. DEFINIREA TIPURILOR DE DATE

Limbajul PASCAL oferă utilizatorului tipurile predefinite de date `integer`, `real`, `boolean`, `char` ş.a. Dacă e necesar, programatorul poate defini tipuri proprii de date, de exemplu, *enumerare* şi *subdomeniu*.

Denumirea unui tip de date şi mulţimea lui de valori se definesc cu ajutorul următoarelor unităţi gramaticale:

<Tipuri> ::= **type** *<Definiţie tip>*; { *<Definiţie tip>* ; }
<Definiţie tip> ::= *<Identificator>* = *<Tip>*
<Tip> ::= *<Identificator>* | *<Tip enumerare>* |
<Tip subdomeniu> | *<Tip tablou>* |
<Tip articol> | *<Tip mulţime>* |
<Tip fişier> | *<Tip referinţă>*
<Tip enumerare> ::= (*<Identificator>* {, *<Identificator>*})
<Tip subdomeniu> ::= *<Constantă>* .. *<Constantă>*

Diagramele sintactice corespunzătoare sînt prezentate în *fig. 2.2.*

Exemple:

```
1) type  T1=(A, B, C, D, E, F, G, H);
         T2=B..F;
         T3=C..H;
2) type  Pozitiv=1..MaxInt;
         Natural=0..MaxInt;
         Negativ=-MaxInt..-1;
3) type  Abatere=-10...+10;
         Litera='A'..'Z';
         Cifra='0'..'9';
```

Clasificarea tipurilor de date ale limbajului PASCAL este prezentată în *fig. 2.3*. Tipurile studiate deja sînt prezentate pe un fundal gri.

În anumite construcţii ale limbajului PASCAL variabilele şi constantele trebuie să fie de tipuri identice sau compatibile.

Două tipuri sînt **identice,** dacă ele au fost definite cu acelaşi nume de tip.

De exemplu, fie

```
type  T4=integer;
      T5=integer;
```

Aici tipurile `integer`, `T4` şi `T5` sînt identice.

Două tipuri sînt identice şi atunci cînd sînt definite cu nume diferite, dar aceste nume sînt echivalente prin tranzitivitate.

De exemplu, fie

```
type  T6=real;
      T7=T6;
      T8=T7;
```

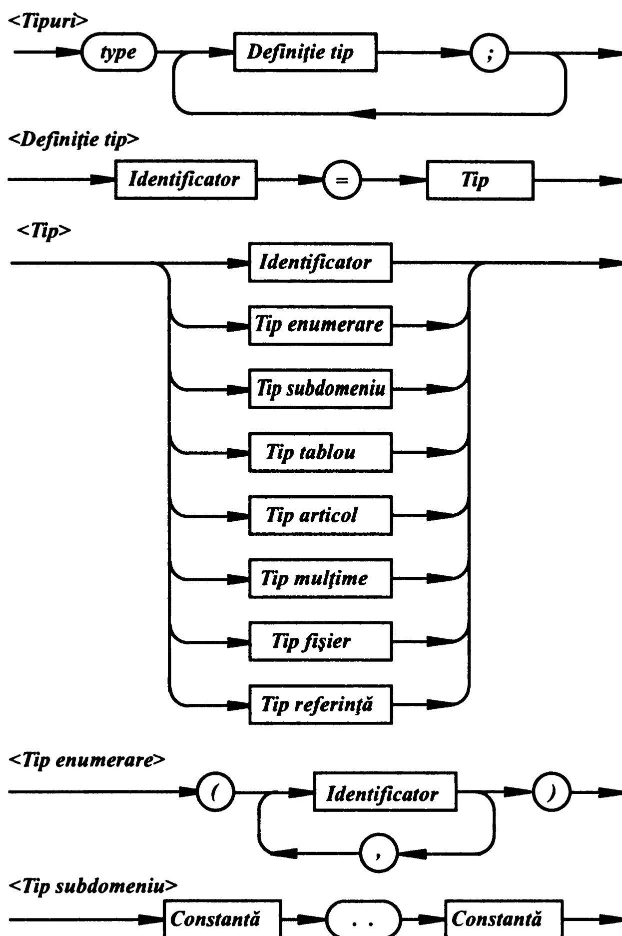


Fig. 2.2. Diagramele sintactice pentru definirea tipurilor de date

Aici `real`, `T6`, `T7` şi `T8` sînt tipuri identice.

Două tipuri sînt **compatibile** atunci cînd este adevărată cel puţin una din următoarele afirmaţii:

- cele două tipuri sînt identice;
- un tip este un subdomeniu al celuilalt tip;
- ambele tipuri sînt subdomenii ale aceluiaşi tip de bază.

De exemplu, în prezenţa declaraţiilor

```
Type  Zi=(L, Ma, Mi, J, V, S, D);
      ZiDeLucru=(L, Ma, Mi, J, V);
      ZiDeOdihna=(S, D);
      Culoare=(Galben, Verde, Albastru, Violet);
```

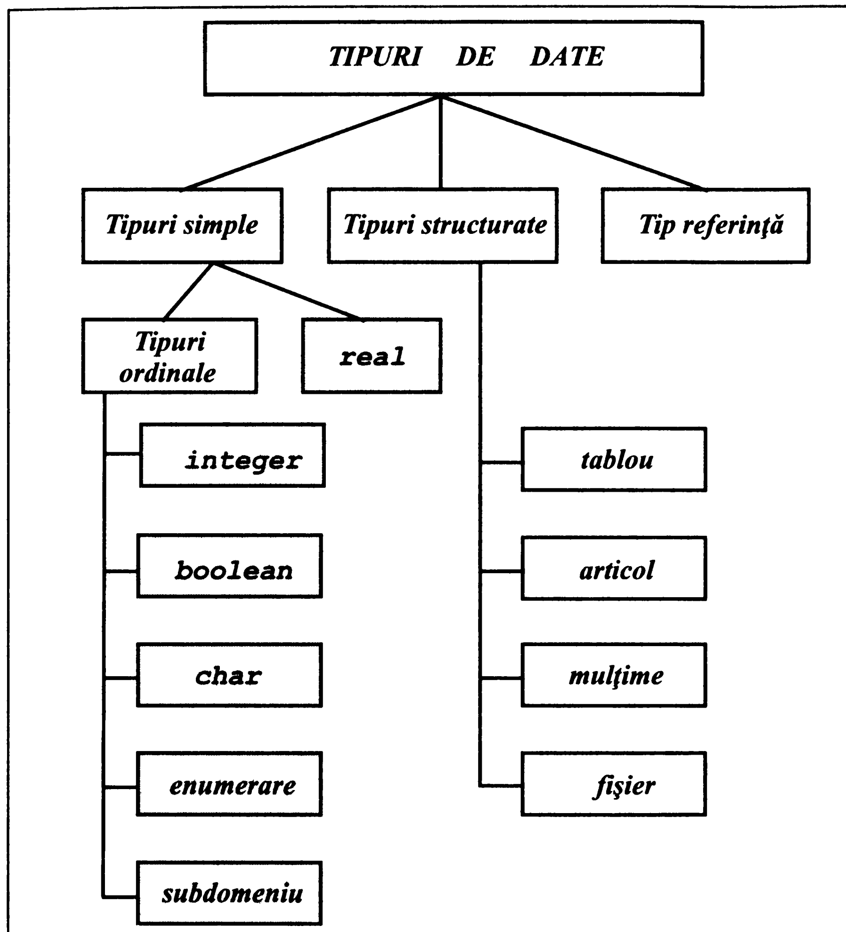


Fig. 2.3. Clasificarea tipurilor de date

tipurile `Zi`, `ZiDeLucru`, `ZiDeOdihna` sînt compatibile. Tipurile `Zi` și `Culoare` sînt tipuri incompatibile. Prin urmare, sînt admise operațiile relaționale

```
L<D
Mi<>D
Verde<>Violet
```

etc. și nu sînt admise operațiile de tipul:

```
L<Violet
Verde=V
S<>Albastru
```

ş.a.m.d.

În completare la tipurile de date definite de utilizator explicit cu ajutorul cuvîntului-cheie **type**, într-un program PASCAL pot fi definite şi tipuri anonime (fără denumire).

Un **tip anonim** se defineşte implicit într-o declaraţie de variabile.

Exemplu:

```
var i : 1..20;
    s : (Alfa, Beta, Gama, Delta);
    t : Alfa..Gama;
```

Se observă că tipul subdomeniu `1..20`, tipul *enumerare* `(Alfa, Beta,Gama,Delta)` şi tipul *subdomeniu* `Alfa..Gama` nu au denumiri proprii.

De regulă, tipurile anonime se utilizează în programele cu un număr mic de variabile.

Întrebări şi exerciţii

➊ Se consideră următorul program:

```
Program P28;
  type T1=-100..100;
       T2='A'..'H';
       T3=(A, B, C, D, E, F, G, H);
       T4=A..E;
       T5=integer;
       T6=real;
       T7=char;
       T8=boolean;
  var i : T1;    j : T5;
      k : T2;    m : T3;
      n : T4;    p : real;
      q : T6;    r : char;
      s : T7;    t : boolean;
      z : T8;    y : real;
  begin
        { ... calcule ce utilizează }
        { variabilele în studiu ... }
        writeln('Sfîrşit');
  end.
```

Precizaţi tipurile de date ale programului. Ce valori poate lua fiecare variabilă din acest program? Care tipuri sînt compatibile?

❷ Indicaţi pe diagramele sintactice din *fig. 2.2* drumurile care corespund definiţiilor tipurilor de date din programul P28.

❸ Precizaţi tipurile anonime de date din următorul program:

```
Program P29;
  type T1=integer;
       T2=-150..150;
       T3=1..5;
  var i : T1;
      j : T2;
      k : T3;
      m : 1..5;
      n : (Unu, Doi, Trei, Patru, Cinci);
  begin
    { ... calcule ce utilizează }
    { variabilele în studiu ... }
    writeln('Sfîrşit')
  end.
```

Ce valori poate lua fiecare variabilă din acest program?

❹ Se consideră următoarele declaraţii:

```
type T1=boolean;
     T2=T1;
     T3=T2;
     T4=T3;
var  x:T4;
```

Precizaţi valorile pe care le poate lua variabila x şi operaţiile tipului corespunzător de date.

❺ Cînd două tipuri de date sînt identice? Daţi exemple.

❻ Cînd două tipuri de date sînt compatibile? Daţi exemple.

❼ Se consideră declaraţiile:

```
type T1=integer;
     T2=T1;
     T3=-5..+5;
     T4=T3;
     T5=-10..+10;
     T6=(A,B,C,D,E,F,G,H);
     T7=A..D;
     T8=E..H;
     T9='A'..'D';
     T10='E' ..'H';
```

Precizaţi tipurile identice şi tipurile compatibile de date.

2.10. DECLARAŢII DE VARIABILE

Cunoaştem deja că fiecare variabilă care apare într-un program PASCAL în mod obligatoriu se asociază cu un anumit tip de date. Pentru aceasta se utilizează următoarele construcţii gramaticale:

<*Variabile*> ::=
  var <*Declaraţie variabile*> ; {<*Declaraţie variabile*>;}

<*Declaraţie variabile*> ::=
  <*Identificator*> {, <*Identificator*>} : <*Tip*>

Diagramele sintactice ale unităţilor gramaticale în studiu sînt prezentate în *fig. 2.4*.

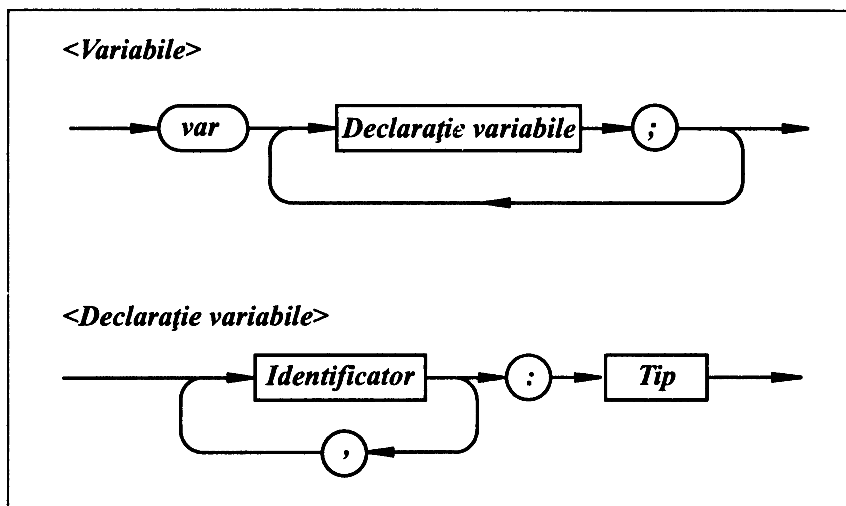


Fig. 2.4. Diagramele sintactice <*Variabile*> şi <*Declaraţie variabile*>

Declaraţiile de variabile pot utiliza tipuri predefinite de date (`integer`, `real`, `char`, `boolean` ş.a.) şi tipuri definite de utilizator (*enumerare, subdomeniu* etc.).

Exemple:

```
1) var  i, j : integer;
        x : real;
        p : boolean;
        r, s : char;
2) type T1=(A, B, C, D, E, F, G, H);
        T2=C..F;
        T3=1..10;
        T4='A'..'Z';
```

```
   var  x, y, z : real;
        r, s : char;
        i, j : integer;
        k : T3;
        p : T1;
        q : T2;
        v : T4;
3) type Zi=(L, Ma, Mi, J, V, S, D);
   var  x, y : real;
        z : Zi;
        z1 : L..V;
        m, n : 1..10;
```

Menţionăm că în ultimul exemplu tipul variabilelor `z1`, `m` şi `n` este definit direct în declaraţiile de variabile. Prin urmare, variabilele în studiu aparţin unor tipuri anonime de date.

Întrebări şi exerciţii

❶ Determinaţi tipul variabilelor din următorul program:

```
Program P30;
type T1=integer;
     Studii=(Elementare, Medii, Superioare);
     T2=real;
     Culoare=(Galben, Verde, Albastru, Violet);
var x : real;
    y : T1;
    i : integer;
    j : T2;
    p : boolean;
    c : Culoare;
    s : Studii;
    q : Galben..Albastru;
    r : 1..9;
begin
   { ... calcule în care se utilizează }
   { variabilele în studiu ... }
   writeln('Sfîrşit');
end.
```

Precizaţi valorile pe care le poate lua fiecare variabilă şi operaţiile tipului corespunzător de date.

❷ Indicaţi pe diagramele sintactice din *fig. 2.4* drumurile care corespund declaraţiilor de variabile din programul `P30`.

❸ Ce mesaje vor fi afişate pe ecran pe parcursul compilării următorului program?

```
Program P31;
var i, j : integer;
begin
```

```
i:=1; j:=2; k:=i+j;
  writeln('k=', k);
end.
```

❹ Cum se declară variabilele unui tip anonim de date?

2.11. DEFINIŢII DE CONSTANTE

Se ştie că valorile unui tip de date pot fi referite prin variabile şi constante. Pentru a face programele mai lizibile şi uşor de modificat, limbajul PASCAL permite reprezentarea constantelor prin denumiri simbolice. Identificatorul care reprezintă o constantă se numeşte *nume de constantă* sau, pur şi simplu, *constantă*. Peste tot în program, unde apare un astfel de nume, el va fi înlocuit cu valoarea corespunzătoare.

Definirea numelor de constante se face cu ajutorul următoarelor construcţii gramaticale:

<*Constante*> ::=
    const <*Definiţie constantă*>; { <*Definiţie constantă*>;}
<*Definiţie constantă*> ::= <*Identificator*> = <*Constantă*>
<*Constantă*> ::=
    [+ | –] <*Număr fără semn*> |
    [+ | –] <*Nume de constantă*> |
    <*Şir de caractere*>

Diagramele sintactice ale unităţilor gramaticale în studiu sînt prezentate în *fig. 2.5.*

Exemple:

```
1)  const a=10;
          b=9.81;
          c='*';
          t='TEXT';
2)  const NumarCaractere=60;
          LungimePagina=40;
3)  const n=10;
          m=20;
          Pmax=2.15e+8;
          Pmin=-Pmax;
          S='STOP';
```

Spre deosebire de variabile, tipul cărora se indică explicit în declaraţiile de variabile, tipul constantelor se indică implicit prin forma lor textuală. În exemplele de mai sus tipul constantelor este:

a, NumarCaractere, LungimePagina, n, m — integer;
b, Pmax, Pmin — real;

`c` — `char`;
`t`, `S` — şir de caractere.

În programul `P32` se definesc constantele `Nmax`, `Nmin`, `Pi`, `Separator`, `Indicator` şi `Mesaj`. Valorile acestor constante se afişează pe ecran.

```
Program P32;
  { Definiţii de constante }
  const Nmax=40;          { Constantă de tip integer }
      Nmin=-Nmax;         { Constantă de tip integer }
      Pi=3.14;            { Constantă de tip real }
      Separator='\';      { Constantă de tip char }
      Indicator=TRUE;     { Constantă de tip boolean }
      Mesaj='Verificaţi imprimanta';
                          { Şir de caractere }
begin
  writeln(Nmax);
  writeln(Nmin);
  writeln(Pi);
  writeln(Separator);
  writeln(Indicator);
  writeln(Mesaj);
  { ... calcule în care se utilizează }
  { constantele în studiu ... }
  end.
```

În mod obişnuit, datele constante ale unui program, de exemplu, numărul de linii ale unui tabel, numărul π, acceleraţia căderii libere *g* ş.a.m.d., se includ în definiţii de constante. Acest fapt permite modificarea constantelor, fără a schimba programul în întregime.

În programul `P33` lungimea *L* şi aria cercului *S* se calculează conform formulelor:

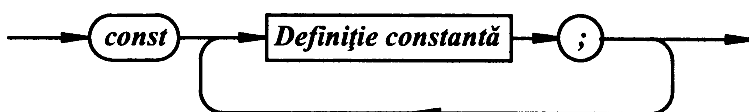
$$L = 2\pi r\ ;$$
$$S = \pi r^2,$$

unde *r* este raza cercului. Numărul π este reprezentat prin constanta `Pi = 3.14`.

```
Program P33;
  { Lungimea şi aria cercului }
  const Pi=3.14;
  var L, S , r :real;
  begin
    writeln('Introduceţi raza r:');
    readln(r);
    L:=2*Pi*r;
    writeln('Lungimea L=', L);
    S:=Pi*r*r;
```

```
    writeln('Aria S=', S);
  end.
```

<Constante>



<Definiţie constantă>



<Constantă>

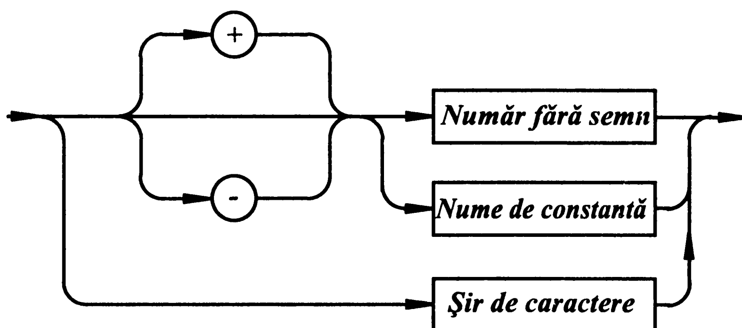


Fig. 2.5. Diagramele sintactice *<Constante>*, *<Definiţie constantă>* şi *<Constantă>*

Dacă utilizatorul are nevoie de rezultate mai exacte, se modifică numai linia a treia a programului `P33`:

```
const Pi=3.141592654;
```

Evident, restul programului rămîne neschimbat.

Spre deosebire de variabile, valorile constantelor nu pot fi modificate prin atribuire sau operaţii de citire. Nerespectarea acestei reguli va declanşa erori de compilare.

Întrebări şi exerciţii

❶ Determinaţi tipul următoarelor constante:

```
a)const a=29.1;      b) const d=-16.82e-14;
        b=TRUE;              f=-d;
        c=18;                i=15;
c) const t='F';      d) const x=65;
        q=FALSE;             y=-x;
        m='PAUZĂ';           z=-y;
```

❷ Elaboraţi un program care afişează pe ecran valorile constantelor din exerciţiul 1.

❸ Indicaţi pe diagramele sintactice din *fig. 2.5* drumurile care corespund definiţiilor de constante din programul P32.

❹ Inseraţi între cuvintele-cheie **begin** şi **end** ale programului P32 una dintre liniile ce urmează:

```
Nmax:=10;
readln(Nmax);
```

Explicaţi mesajele afişate pe ecran pe parcursul compilării programului modificat.

❺ Se consideră programul:

```
Program P34;
  const t='1'; s=-t;
  begin
    writeln(s);
  end.
```

Ce mesaje vor fi afişate pe ecran în procesul compilării?

❻ Ce rezultate se vor afişa după execuţia următorului program:

```
Program P35;
  const i=1; j=i; k=j;
  var x, y, z : integer;
  begin
    x:=i+1; writeln(x);
    y:=j+2; writeln(y);
    z:=k+3; writeln(z);
  end.
```

❼ Se consideră programul:

```
Program P36;
  const a=1; b=2; c=3; d=4;
  var i, j, k : integer;
```

```
begin
  i:=a+1; writeln(i);
  j:=b+1; writeln(j);
  k:=c+1; writeln(k);
  d:=a+1; writeln(d);
end.
```

Explicaţi mesajele afişate pe ecran.

❽ Excludeţi din programul ce urmează linia care conţine o eroare.

```
Program P37;
  const a=1;
  var i, j : integer;
  begin
    writeln('Introduceţi i=');
    readln(i); j:=i+a;
    writeln(j);
    writeln('Introduceţi a=');
    readln(a);
    j:=i+a;
    writeln(j);
  end.
```

Ce rezultate vor fi afişate pe ecran după execuţia programului modificat?

Capitolul 3

INSTRUCŢIUNI

3.1. CONCEPTUL DE ACŢIUNE

Un program PASCAL constă din două părţi: partea declarativă şi partea executabilă. În partea declarativă se descriu datele cu care va opera programul, iar în partea executabilă se descriu acţiunile (operaţiile) cu aceste date.

Acţiunile necesare pentru a prelucra datele unui program şi ordinea executării lor se definesc cu ajutorul **instrucţiunilor**. Există două categorii de instrucţiuni:

1) instrucţiuni simple;
2) instrucţiuni structurate.

Instrucţiunile simple nu conţin alte instrucţiuni. Instrucţiunile simple sînt:

- instrucţiunea de atribuire;
- instrucţiunea de apel de procedură;
- instrucţiunea de salt necondiţionat;
- instrucţiunea de efect nul.

Instrucţiunile structurate sînt construite din alte instrucţiuni. Instrucţiunile structurate sînt:

- instrucţiunea compusă;
- instrucţiunile condiţionale **`if`** şi **`case`**;
- instrucţiunile iterative **`for`**, **`while`** şi **`repeat`**;
- instrucţiunea **`with`**.

În cadrul unui program instrucţiunile pot fi prefixate de etichete. Etichetele pot fi referite în instrucţiunile de salt necondiţionat **`goto`**. Amintim că eticheta este un număr întreg fără semn (vezi paragraful 1.10).

Diagrama sintactică *<Instrucţiune>* este prezentată în *fig. 3.1*. Menţionăm că eticheta se separă de instrucţiunea propriu-zisă prin simbolul "`:`" (două puncte).

Într-un program PASCAL scrierea instrucţiunilor pe linii nu este limitată, o instrucţiune poate ocupa una sau mai multe linii, sau într-o linie pot fi mai multe instrucţiuni. Ca separator de instrucţiuni se foloseşte simbolul "`;`" (punct şi virgulă).

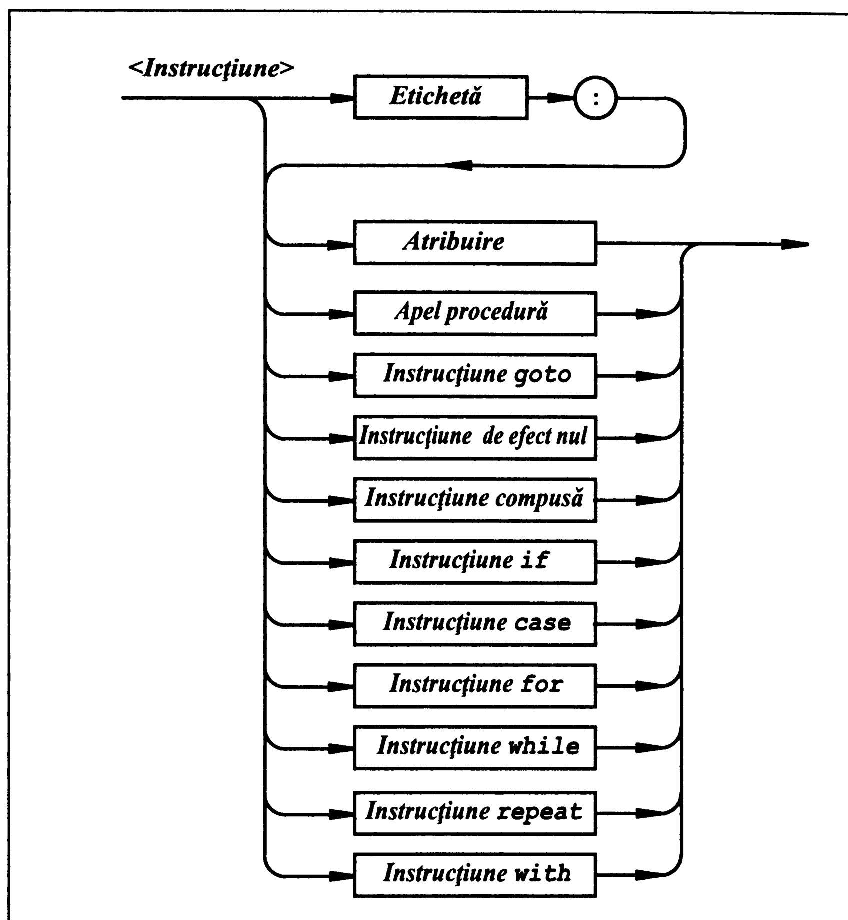


Fig. 3.1. Diagrama sintactică *<Instrucţiune>*

3.2. EXPRESII

Formulele pentru calculul unor valori se reprezintă în PASCAL prin **expresii**. Acestea sînt formate din operanzi (constante, variabile, referinţe de funcţii) și operatori (simbolurile operaţiilor). Operatorii se clasifică după cum urmează:

<Operator multiplicativ> ::= * | / | **div** | **mod** | **and**
<Operator aditiv> ::= + | – | **or**
<Operator relaţional> ::= < | <= | = | >= | > | <> | **in**

În componenţa expresiilor intră factori, termeni și expresii simple.

Factorul poate fi o variabilă, o constantă fără semn, apelul unei funcţii ş.a. Mai exact,

<Factor> ::= *<Variabilă>* | *<Constantă fără semn>* | <Apel funcţie> | **not** *<Factor>* | (*<Expresie>*) | *<Constructor mulţime >*

Exemple:

```
15
x
p
sin(x)
not p
```

Un **termen** are forma:

<Termen> ::= *<Factor>* {*<Operator multiplicativ>* *<Factor>*}

Exemple:

```
15
x
15*x
x*y*z
p and q
sin(x)/3
```

Prin **expresie simplă** se înţelege:

<Expresie simplă>::=[+ | –] *<Termen>* {*<Operator aditiv>* *<Termen>*}

Exemple:

```
+15
-x
15*x+sin(x)/3
p or q
```

La rîndul său, o **expresie** are forma:

<Expresie> ::=
<Expresie simplă> {*<Operator relaţional>* *<Expresie simplă>*}

Exemple:

```
-15
x*y+cos(z)/4
x<15
15*x+sin(x)/3<>11
x+6>z-3.0
```

Diagramele sintactice ale unităţilor gramaticale în studiu sînt prezentate în *fig. 3.2* şi *3.3*.

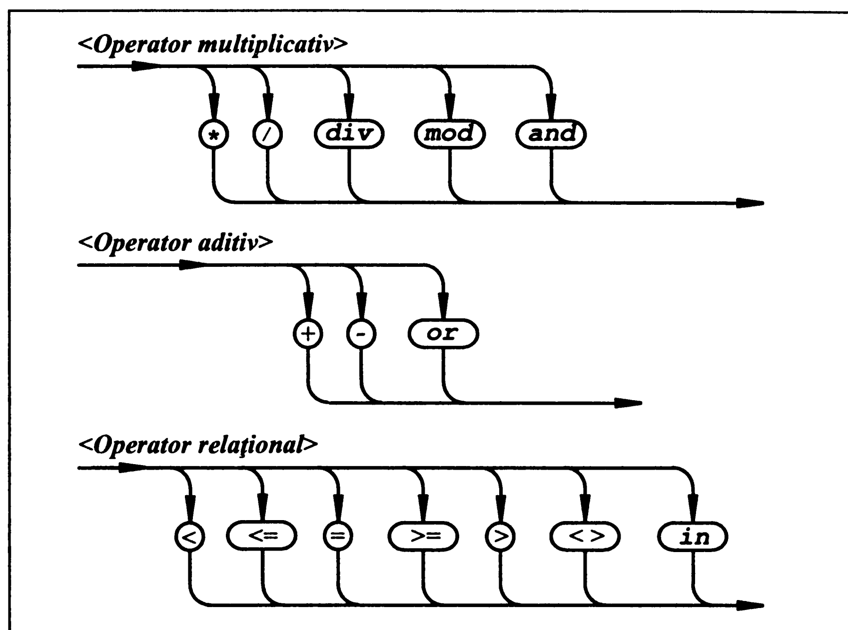


Fig. 3.2. Diagramele sintactice pentru operatori

O expresie luată în paranteze se transformă într-un factor. Cu astfel de factori se pot forma noi termeni, expresii simple, expresii ş.a.m.d.

Exemple:

notaţia matematică	*notaţia în PASCAL*
$\frac{a+b}{c+d}$	`(a+b)/(c+d)`
$\frac{-b+sin(b-c)}{2a}$	`(-b+sin(b-c))/(2*a)`
$-\frac{1}{xy}$	`-1/(x*y)`
$\underline{p<q}$ & $r>s$	`(p<q)and(r>s)`
$x \vee y$	`not (x or y)`
$\frac{1}{a+b} > \frac{1}{c+d}$	`1/(a+b)>1/(c+d)`

În cadrul unei expresii un apel de funcţie poate să apară peste tot unde apare o variabilă sau o constantă (*fig. 3.3*). Limbajul PASCAL conţine un set de **funcţii predefinite**, cunoscute oricărui program. Aceste funcţii sînt prezentate în *tabelul 3.1*.

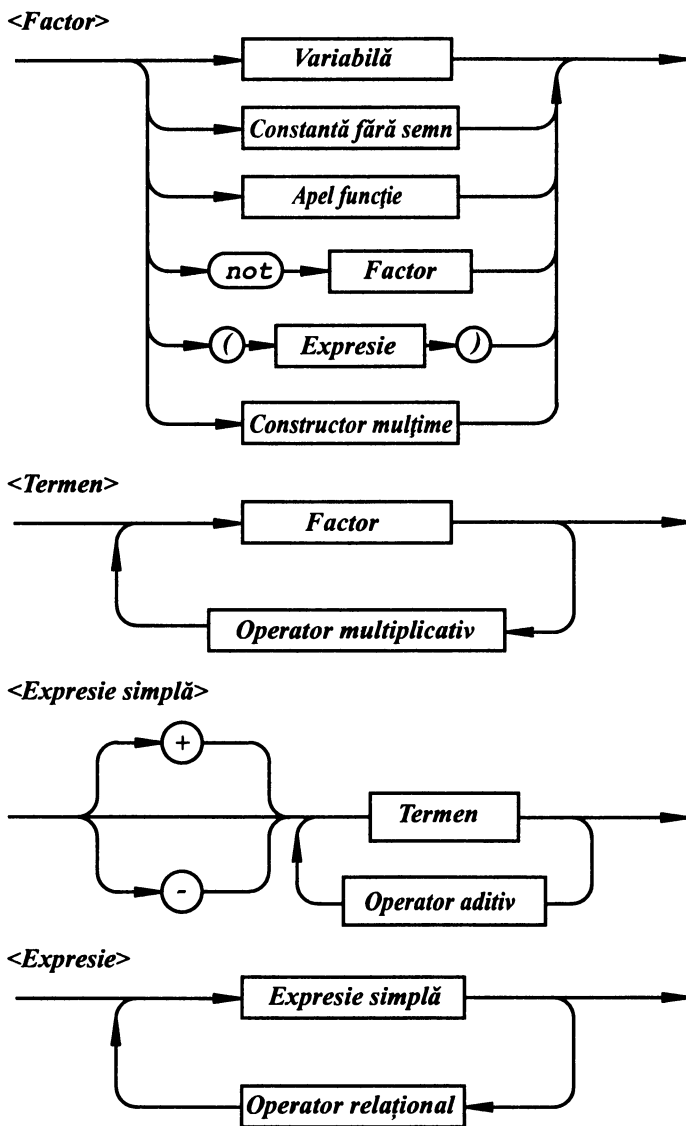


Fig. 3.3. Diagramele sintactice pentru definirea expresiilor

Tabelul 3.1. Funcţiile predefinite ale limbajului PASCAL

DENUMIREA FUNCŢIEI	*NOTAŢIA ÎN PASCAL*
valoarea absolută $\|x\|$	`abs(x)`
sinus $\sin x$	`sin(x)`
cosinus $\cos x$	`cos(x)`
arctangenta $\text{arctg}\, x$	`arctan(x)`
pătratul lui x x^2	`sqr(x)`
rădăcina pătrată $\sqrt{x}$	`sqrt(x)`
puterea numărului e e^x	`exp(x)`
logaritmul natural $\ln x$	`ln(x)`
rotunjirea lui x	`round(x)`
trunchierea lui x	`trunc(x)`
paritatea numărului i (*false* pentru i par şi *true* în caz contrar)	`odd(i)`
numărul valorii ordinale v	`ord(v)`
predecesorul lui v	`pred(v)`
succesorul lui v	`succ(v)`
caracterul cu numărul i	`chr(i)`
testarea sfîrşitului de fişier	`eof(f)`
testarea sfîrşitului de linie	`eoln(f)`

Întrebări şi exerciţii

❶ Scrieţi conform regulilor limbajului PASCAL următoarele expresii:

a) $a^2 + b^2$;

b) $a^2 + 2ab + b^2$;

c) $(a + b)^2$;

d) $v_0 t + \frac{a t^2}{2}$;

e) $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$;

f) $\cos\alpha + \cos\beta$;

g) $\cos(\alpha + \beta)$;

h) $2\pi r$;

i) πr^2;

j) $x_1 x_2 \vee x_3 x_4$;

k) $\overline{x_1 \vee x_2}$;

l) $|x| < 3$;

m) $z < 6 \,\&\, |q| > 13{,}4$;

n) $x > 0 \,\&\, y > 8 \,\&\, R < 15$.

❷ Indicaţi pe diagramele sintactice din *fig. 3 3* drumurile care corespund următoarelor expresii PASCAL:

a) `x`

b) `3.14`

c) `sin(x)`

d) **`not`** `q`

e) `(+3.14)`
f) `x>2.85`
g) `sqr(b)-4*a*c>0`
h) `(a>b) and (c>d)`

❸ Transpuneţi expresiile PASCAL în notaţii obişnuite:

a) `sqr(a)+sqr(b)`
b) `2*a* (b+c)`
c) `sqrt((a+b)/(a-b))`
d) `exp(x+y)`
e) `cos(ALFA-BETA)`
f) `sqr(a+b)/(a-b)`
g) `x>0 or q<p`
h) `not(x and y)`

❹ Care din expresiile PASCAL ce urmează sînt greşite? Pentru a argumenta răspunsul, utilizaţi diagramele sintactice din *fig. 3.3.*

a) `((((+x))))`
b) `((((x))))`
c) `sinx+cosx`
d) `sqr(x)+sqr(y)`
e) `a<<b or c>d`
f) `not not not p`
g) `q and not p`
h) `not q and p`
i) `a+-b`
j) `sin(-x)`
k) `sin-x`
l) `cos(x+y)`
m) `sin(abs(x)+abs(y))`
n) `sqrt(-y)`

3.3. EVALUAREA EXPRESIILOR

Prin evaluarea unei expresii se înţelege calculul valorii ei. Rezultatul furnizat depinde de valorile operanzilor şi de operatorii care acţionează asupra acestora.

Regulile de evaluare a unei expresii sînt cele obişnuite în matematică:

– operaţiile se efectuează conform priorităţii operatorilor;

– în cazul priorităţilor egale operaţiile se efectuează de la stînga spre dreapta;

– mai întîi se calculează expresiile dintre paranteze.

Priorităţile operatorilor sînt indicate în *tabelul 3.2.*

Tabelul 3.2. Priorităţile operaţiilor limbajului PASCAL

CATEGORIE	*OPERATORI*	*PRIORITATE*
operatori unari	not, @	prima (cea mai mare)
operatori multiplicativi	*, /, div, mod, and	a doua
operatori aditivi	+, -, or	a treia
operatori relaţionali	<, <=, =, >=, >, <>, in	a patra (cea mai mică)

Exemplu:

Fie x=2 şi y=6. Atunci:
a) 2*x+y = 2·2+6=4+6=10;
b) 2*(x+y) = 2·(2+6)=2·8=16;
c) x+y/x-y = 2+6/2–6=2+3–6=5–6= –1;
d) (x+y)/x-y = (2+6)/2– 6=8/2–6=4–6= –2;
e) x+y/(x-y) = 2+6/(2– 6)=2+6/(–4)=2+(–1,5)=0,5;
f) x+y<15 = 2+6<15=8<15=true;
g) (x+y<15) and (x>3) = (2+6<15) and (2>3) = (8<15) and 2>3) = true and false = false.

Se observă că părţile componente ale unei expresii (*fig. 3.3*) se calculează în următoarea ordine:

1) factorii;
2) termenii;
3) expresiile simple;
4) expresia propriu-zisă.

Valoarea curentă a unei expresii poate fi afişată pe ecran cu ajutorul procedurii writeln:

writeln(<*Expresie*>)

Programul P38 afişează pe ecran rezultatele evaluării expresiilor x*y+z şi x+y<z-1.0. Valorile curente ale variabilelor x, y şi z sînt citite de la tastatură.

```
Program P38;
  { Evaluarea expresiilor }
  var x, y, z : real;
  begin
    writeln('Introduceţi numerele reale x, y, z:');
    readln(x, y, z);
    writeln(x*y+z);
    writeln(x+y<z-1.0);
  end.
```

Întrebări şi exerciţii

❶ Fie x=1, y=2 şi z=3. Evaluaţi următoarele expresii:

a) x+y+2*z;
b) (x+y+2)*z;
c) x*y+y*z;
d) x*(y+y)*z;
e) (x*y+y)*z;
f) x*(y+y*z);
g) x*y<y*z;
h) (x>y) **or** (6*x>y+z);
i) **not** (x+y+z>0);
j) **not** (x+y>0) **and** **not** (z<0).

❷ Care sînt regulile de evaluare a unei expresii PASCAL?
❸ Indicaţi prioritatea fiecărui operator al limbajului PASCAL.
❹ Precizaţi ordinea în care se calculează componentele unei expresii PASCAL.
❺ Elaboraţi un program care evaluează expresiile *c* şi *g* din exerciţiul 1. Valorile curente ale variabilelor reale x, y şi z se citesc de la tastatură.

3.4. TIPUL EXPRESIILOR PASCAL

În funcţie de mulţimea valorilor pe care le poate lua, fiecare expresie se asociază cu un anumit tip de date. Conform conceptului de dată realizat în limbajul PASCAL, **tipul expresiei** derivă (rezultă) din tipul operanzilor şi operatorilor care acţionează asupra acestora. Prin urmare, tipul unei expresii poate fi dedus fără a calcula valoarea ei.

Tipul rezultatelor furnizate de operatori este indicat în *tabelul 3.3*. *Tabelul 3.4* conţine tipul rezultatelor furnizate de funcţiile predefinite ale limbajului PASCAL.

Tabelul 3.3. Tipul rezultatelor furnizate de operatori

OPERATOR	*TIPUL OPERANZILOR*	*TIPUL REZULTATULUI*
`+, -, *`	`integer integer`	`integer`
	unul `integer`, altul `real`	`real`
`/`	`integer` sau `real`	`real`
`div`	`integer integer`	`integer`
`mod`	`integer integer`	`integer`
`not, and, or`	`boolean boolean`	`boolean`
`<, <=, =, >=, >, <>`	tipuri identice	`boolean`
	tipuri compatibile	`boolean`
	unul `integer`, altul `real`	`boolean`

Indiferent de tipul operanzilor, operatorul / (împărţirea) furnizează numai rezultate de tip `real`, iar operatorii relaţionali — numai rezultate de tip `boolean`.

Pentru a afla tipul unei expresii, factorii, termenii şi expresiile simple se examinează în ordinea evaluării lor. Tipul fiecărei părţi componente se deduce cu ajutorul *tabelelor 3.3* şi *3.4*.

De exemplu, fie expresia:

```
(x>i) or (6*i<sin(x/y)),
```

unde `i` este de tipul `integer`, iar `x` şi `y` de tipul `real`.

Aflăm tipul fiecărei părţi componente şi al expresiei în ansamblu în ordinea de evaluare:

1) `x>i` `boolean;`
2) `6*i` `integer;`
3) `x/y` `real;`
4) `sin(x/y)` `real;`
5) `6*i<sin(x/y)` `boolean;`
6) `(x>i) or (6*i<sin(x/y))` `boolean.`

Tabelul 3.4. Tipul rezultatelor furnizate de funcţiile predefinite

FUNCŢIA	*TIPUL ARGUMENTULUI*	*TIPUL REZULTATULUI*
`abs(x)`	`integer` sau `real`	coincide cu tipul lui *x*
`sin(x)`	`integer` sau `real`	`real`
`cos(x)`	`integer` sau `real`	`real`
`arctan(x)`	`integer` sau `real`	`real`
`sqr(x)`	`integer` sau `real`	coincide cu tipul lui *x*
`sqrt(x)`	`integer` sau `real`	`real`
`exp(x)`	`integer` sau `real`	`real`
`ln(x)`	`integer` sau `real`	`real`
`round(x)`	`real`	`integer`
`trunc(x)`	`real`	`integer`
`odd(i)`	`integer`	`boolean`
`ord(v)`	ordinal	`integer`
`pred(v)`	ordinal	coincide cu tipul lui *v*
`succ(v)`	ordinal	coincide cu tipul lui *v*
`chr(i)`	`integer`	`char`
`eof(f)`	fişier	`boolean`
`eoln(f)`	fişier	`boolean`

Prin urmare, expresia în studiu este de tip `boolean`.

Întrucît în procesul deducţiei valorile concrete ale expresiilor în studiu nu se calculează, tipurile subdomeniu se extind la tipurile de bază.

De exemplu, în prezenţa declaraţiilor

```
Type    T1=1..10;  { subdomeniu de integer}
      T2=11..20; { subdomeniu de integer}
var     i:T1;
      j:T2;
```

avem:

<u>*expresie*</u>	<u>*tipul expresiei*</u>
`i+j`	`integer`
`i mod j`	`integer`
`i/j`	`real`
`sin(i+j)`	`real`
`i>j`	`boolean`

ş.a.m.d.

În funcţie de tipul expresiei distingem:
– expresii aritmetice (`integer` sau `real`);
– expresii ordinale (`integer`, `boolean`, `char`, enumerare);
– expresii booleene (`boolean`).

De obicei, expresiile aritmetice se utilizează în calcule (instrucţiunea de atribuire), expresiile ordinale — în instrucţiunile **case** şi **for**, iar expresiile booleene — în instrucţiunile **if, repeat** şi - **while**.

Întrebări şi exerciţii

❶ Prin ce metodă se află tipul unei expresii PASCAL?

❷ În prezenţa declaraţiilor:

```
var x, y : real;
    i, j : integer;
    p, q : boolean;
    r : char;
    s : (A, B, C, D, E, F, G, H);
```

aflaţi tipul următoarelor expresii:

a) `i mod 3;`
b) `i/3;`
c) `i mod 3 > j div 4;`
d) `x+y/(x-y);`
e) `not(x<i);`
f) `sin(abs(i)+abs(j));`
g) `sin(abs(x)+abs(y));`
h) `p and (cos(x)<=sin(y));`
i) `sqr(i)-sqr(j);`
j) `sqr(x)-sqr(y);`
k) `trunc(x)+trunc(y);`
l) `chr(i);`
m) `ord(r);`
o) `ord(s)>ord(r);`
p) `pred(E);`
q) `(-x+sin(x-y))/(2*i).`

❸ Tipul unei expresii poate fi aflat din forma textuală a rezultatelor afişate pe ecran de instrucţiunea

writeln(<*Expresie*>)

Exemple:

rezultatul afişat pe ecran	*tipul expresiei*
100	integer
1.0000000000E+02	real
true	boolean

Elaboraţi programele respective şi precizaţi tipul expresiilor ce urmează, pornind de la forma textuală a rezultatelor afişate:

a) `1+1.0;`
b) `1/1+1;`
c) `9*3 mod 4;`
d) `4*x>9*y;`
e) `chr(65);`
f) `not(x>y);`
g) `pred(9)>succ(7);`
h) `15 div ord(3);`
i) `trunc(x)+round(6*y);`
j) `sqr(3)-sqrt(16).`

Se consideră că variabilele x şi y sînt de tip real.

❹ Se consideră declaraţiile

```
Type T1=1..10;
     T2=11..20;
     T3='A'..'Z';
     T4=(A, B, C, D, E, F, G, H);
```

```
var   i:T1;
      j:T2;
      k:T3;
      m:'C'..'G';
      n:T4;
      p:C..G;
      q:boolean;
```

Aflaţi tipul următoarelor expresii:

a) `i-j;`
b) `i div j;`
c) `6.3*i;`
d) `cos(3*i-6*j);`
e) `4*i>5*j;`
f) `k<m;`
g) `k<>m;`
h) `chr(i);`
i) `ord(k);`
j) `ord(m);`
k) `n>p;`
l) `ord(n);`
m) `succ(n);`
n) `pred(p);`
o) `ord(p);`
p) `ord(k)>ord(m);`
q) `(i>j) and q;`
r) `not(i+j>0) or q.`

3.5. INSTRUCŢIUNEA DE ATRIBUIRE

Instrucţiunea în studiu are forma:

< *Variabilă*> ::=< *Expresie*>

Execuţia unei instrucţiuni de atribuire presupune:

a) evaluarea expresiei din partea dreaptă;
b) atribuirea valorii obţinute variabilei din partea stîngă.

Exemple:

```
x:=1
y:=x+3
z:=sin(x)+cos(y)
p:=not q
q:=(a<b) or (x<y)
c:='A'
```

De reţinut că simbolul ": =" (se citeşte "atribuire") desemnează o atribuire şi nu trebuie confundat cu operatorul de relaţie "=" (egal).

O atribuire are loc dacă variabila şi rezultatul evaluării expresiei sînt compatibile din punct de vedere al atribuirii. În caz contrar, se va declanşa o eroare.

Variabila şi rezultatul evaluării expresiei sînt **compatibile din punct de vedere al atribuirii** dacă este adevărată una din următoarele afirmaţii:

1) variabila şi rezultatul evaluării sînt de tipuri identice;
2) tipul rezultatului este un subdomeniu al tipului variabilei;

3) ambele tipuri sînt subdomenii ale aceluiaşi tip, iar rezultatul este în subdomeniul variabilei;

4) variabila este de tip `real`, iar rezultatul – de tip `integer` sau un subdomeniu al acestuia.

Pentru exemplificare, să considerăm următorul program:

```
Program P39;
  { Compatibilitate din punctul de vedere al atribuirii}
  type  T1=1..10; { subdomeniu de integer }
        T2=5..15; { subdomeniu de integer }
  var i : T1;
      j : T2;
      k, m, n : integer;
      x : real;
  begin
    write('k='); readln(k);
    i:=k;        { corect pentru 1<=k<=10 }
    write('m='); readln(m);
    j:=m;        { corect pentru 5<=m<=15 }
    write('n='); readln(n);
    i:=n+5;      { corect pentru -4<=n<=5 }
    j:=n+2;      { corect pentru 3<=n<=13 }
    x:=i+j;
    writeln('x=', x);
end.
```

Programul va derula fără erori numai pentru următoarele valori de intrare:

$1 \le k \le 10$;
$5 \le m \le 15$;
$3 \le n \le 5$.

Evident, în programul `P39` instrucţiunile de tipul

```
k:=x;
i:=x+1;
j:=sin(i)
```

ş.a.m.d. ar fi incorecte întrucît `x`, `x+1`, `sin(i)` sînt expresii de tip `real`, iar variabilele `k`, `i`, `j` sînt de tip `integer` sau subdomenii ale acestuia.

Întrebări şi exerciţii

❶ Cum se execută o instrucţiune de atribuire?

❷ Explicaţi termenul "compatibilitate din punct de vedere al atribuirii".

❸ Se consideră următoarele declaraţii:

```
Type  Zi=(L, Ma, Mi, J, V, S, D);
      Culoare=(Galben, Verde, Albastru, Violet);
var i, j, k: integer;
```

```
z: Zi;
c: Culoare;
x: real;
```

Care din instrucţiunile ce urmează sînt corecte?

a) `i:=12;`
b) `j:=ord(i);`
c) `x:=ord(z)+1;`
d) `k:=ord(x)+2;`
e) `c:=i+4;`
f) `c:=Verde;`
g) `z:=D;`
h) `c:=Pred(Galben);`
i) `x:=Succ(z);`
j) `i:=Succ(c).`

❹ Precizaţi pentru care valori ale variabilei `j` programul `P40` va derula fără erori?

```
Program P40;
  var i : -10..+10; j : integer;
  begin
    write('j='); readln(j);
    i:=j+15;
    writeln('i=', i);
  end.
```

3.6. INSTRUCŢIUNEA *APEL DE PROCEDURĂ*

Procedura este un subalgoritm scris în limbaj de programare ce poate fi apelat din mai multe puncte ale unui program. Fiecare procedură are un nume, de exemplu, `readln`, `writeln`, `CitireDate`, `A15` ş.a.m.d. Limbajul PASCAL include un set de proceduri predefinite, cunoscute oricărui program: `read`, `readln`, `write`, `writeln`, `get`, `put`, `new` etc. În completare, programatorul poate defini proceduri proprii.

Instrucţiunea *apel de procedură* lansează în execuţie procedura cu numele specificat. Sintaxa instrucţiunii în studiu este

<*Apel procedură*> ::=
<*Nume procedură*> [<*Listă parametri actuali*>];
<*Nume procedură*> ::= <*Identificator*>
<*Listă parametri actuali*> ::=
(<*Parametru actual*> {, <*Parametru actual*>})

Diagramele sintactice ale formulelor în studiu sînt prezentate în *fig. 3.4*.

În mod obişnuit, <*Parametrul actual*> este o expresie.

Exemple:

```
readln(x);
readln(x, y, z);
```

```
CitireDate(f, t);
Exit;
writeln(x+y, sin(x))
```

Tipul parametrilor actuali şi ordinea în care aceştia apar în listă sînt impuse de declaraţiile procedurii respective. În consecinţă, regulile de formare a listelor de parametri actuali vor fi studiate mai amănunţit în capitolele următoare.

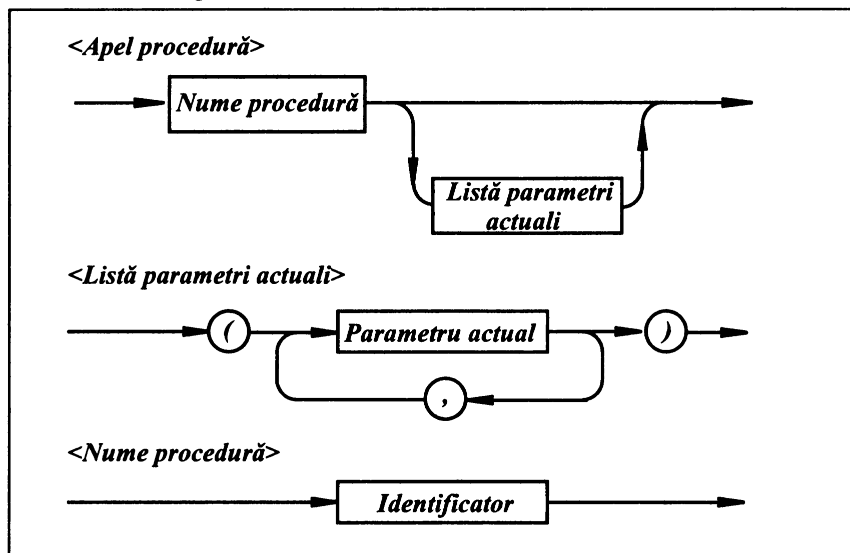


Fig. 3.4. Diagramele sintactice ale instrucţiunii apel de procedură

Întrebări şi exerciţii

❶ Care este destinaţia instrucţiunii *apel de procedură*?

❷ Se consideră următoarele instrucţiuni:

```
readln(x, y, z, q);
CitireDate(ff, tt);
Halt;
writeln('x=', x, 'y=', y);
writeln('x+y=', x+y, 'sin(x)=', sin(x))
```

Precizaţi numele procedurilor apelate, numărul de parametri actuali din fiecare apel şi parametrii propriu–zişi.

❸ Indicaţi pe diagramele sintactice din *fig. 3.4* drumurile ce corespund instrucţiunilor din exerciţiul 2.

3.7. AFIŞAREA INFORMAŢIEI ALFANUMERICE

În versiunile uzuale ale limbajului PASCAL ecranul vizualizatorului este desemnat ca dispozitiv-standard de ieşire. De regulă, ecranul este împărţit în zone convenţionale, numite zone-caracter. De obicei, aceste zone formează 25 de linii — cîte 80 de caractere pe linie. Zona în care va fi afişat caracterul curent este indicată de cursor.

Datele de ieşire ale unui program PASCAL pot fi afişate pe ecran printr-un apel

`write(x)` sau `writeln(x)`.

Apelul

`write(`x_1`, `x_2`, ..., `x_n`)`.

este echivalent cu `write(`x_1`); write(`x_2`); ...; write(`x_n`)`

Parametrii actuali dintr-un apel `write` sau `writeln` se numesc ***parametri de ieşire***. Aceştia pot avea una din formele:

e
e:w
e:w:f

unde *e* este o expresie de tip `integer`, `real`, `boolean`, `char` sau şir de caractere a cărei valoare trebuie afişată; *w* şi *f* sînt expresii de tip `integer`, numite ***specificatori de format***.

Expresia *w* specifică prin valoarea sa numărul minim de caractere ce vor fi folosite la afişarea valorii lui *e*; dacă sînt necesare mai puţin de *w* caractere, atunci forma externă a valorii lui *e* va fi completată cu spaţii la stînga pînă la *w* caractere (*fig. 3.5*).

Specificatorul de format *f* are sens în cazul în care *e* este de tip `real` şi indică numărul de cifre care urmează punctul zecimal în scrierea valorii lui *e* în virgulă fixă, fără factor de scală. În lipsa lui *f* valoarea lui *e* se scrie în virgulă mobilă, cu factor de scală (*fig. 3.6*).

Diferenţa dintre procedurile `write` şi `writeln` constă în faptul că, după afişarea datelor, `write` lasă cursorul în linia curentă, în timp ce `writeln` îl trece la începutul unei linii noi. Utilizarea raţională a apelurilor `write`, `writeln` şi a specificatorilor de format asigură o afişare lizibilă a datelor de ieşire. Dacă pe ecran se afişează mai multe valori, se recomandă ca acestea să fie însoţite de identificatorii respectivi sau de mesaje sugestive.

Exemple:

```
write('Suma numerelor introduse este');
writeln(s:20);
writeln('Suma=', s);
writeln('s=' s);
writeln('x=', x, 'y=':5, y, 'z=':5, z)
```

write (-1234)

```
-1234
```

write (-1234:10)

```
     -1234
```

write ('a')

```
a
```

write ('a':5)

```
    a
```

write ('a', 'b')

```
ab
```

write ('a':5, 'b':5)

```
    a    b
```

Fig. 3.5. Semnificaţia specificatorului de format w

write (-1234.567890)

```
-1.2345678900E+03
```

write (-1234.567890:20)

```
   -1.2345678900E+03
```

write (-1234.567890:20:1)

```
             -1234.6
```

write (-1234.567890:20:4)

```
          -1234.5679
```

Fig. 3.6. Semnificaţia specificatorului de format f

Întrebări şi exerciţii

❶ Care este destinaţia specificatorului de format?

❷ Cum se numesc parametrii actuali ai unui apel de procedură `write` sau `writeln`?

❸ Precizaţi formatul datelor afişate pe ecran de programele ce urmează:

```
Program P41;
 { Afişarea datelor de tip integer }
var i : integer;
begin
  i:=-1234;
  writeln(i);
  writeln(i:1);
  writeln(i:8);
  writeln(i, i);
  writeln(i:8, i:8);
  writeln(i, i, i);
  writeln(i:8, i:8, i:8);
end.
```

```
Program P42;
 { Afişarea datelor de tip real }
var x : real;
  begin
    x:=-1234.567890;
    writeln(x);
    writeln(x:20);
    writeln(x:20:1);
    writeln(x:20:2);
    writeln(x:20:4);
    writeln(x, x, x);
    writeln(x:20, x:20, x:20);
    writeln(x:20:4, x:20:4, x:20:4);
  end.
```

```
Program P43;
 { Afişarea datelor de tip boolean }
var p : boolean;
begin
  p:=false;
  writeln(p);
  writeln(p:10);
  writeln(p, p);
  writeln(p:10, p:10);
end.
```

```
Program P44;
 { Afişarea şirurilor de caractere }
begin
  writeln('abc');
  writeln('abc':10);
  writeln('abc', 'abc');
```

```
  writeln('abc':10, 'abc':10);
end.
```

❹ Elaboraţi un program care afişează pe ecran valorile 1234567890, 123, 123.0 şi true după cum urmează:

```
1234567890
123
123.0
true
1234567890
          123
          123.000
          true
```

3.8. CITIREA DATELOR DE LA TASTATURĂ

În mod obişnuit, tastatura vizualizatorului este desemnată ca **dispozitiv-standard de intrare**. Citirea datelor de la tastatură se realizează prin apelul procedurilor predefinite `read` sau `readln`. Lista parametrilor actuali a unui apel `read` sau `readln` poate să includă variabile de tip `integer`, `real`, `char` şi şir de caractere.

Apelul

read(*x*)

are următorul efect. Dacă variabila x este de tip `integer` sau `real`, atunci este citit întregul şir de caractere care reprezintă valoarea întreagă sau reală. Dacă x este de tip `char`, procedura citeşte un singur caracter.

Apelul

read(x_1, x_2, ..., x_n)

este echivalent cu

read(x_1); read(x_2); ...; read(x_n).

Datele numerice introduse de la tastatură trebuie separate prin spaţii sau caractere sfîrşit de linie. Spaţiile dinaintea unei valori numerice sînt ignorate. Şirul de caractere care reprezintă o valoare numerică se conformează sintaxei constantelor numerice de tipul respectiv. În caz contrar, este semnalată o eroare de intrare-ieşire.

De exemplu, fie programul:

```
Program P45;
 { Citirea datelor numerice de la tastatură }
var i,j : integer;
    x,y : real;
```

```
begin
  read(i, j, x, y);
  writeln('Aţi introdus:');
  writeln('i=', i);
  writeln('j=', j);
  writeln('x=', x);
  writeln('y=', y);
end.
```

în care sînt citite de la tastatură valorile variabilelor `i`, `j`, `x`, `y`. După lansarea programului în execuţie, utilizatorul tastează:

```
1<ENTER>
2<ENTER>
3.0<ENTER>
4.0<ENTER>
```

Pe ecran se va afişa:

```
Aţi introdus:
i=1
j=2
x=3.0000000000E+00
y=4.0000000000E+00
```

Acelaşi efect se va obţine şi la tastarea numerelor într-o singură linie:

```
1 2 3.0 4.0<ENTER>
```

Dacă e necesar, numerele întregi, introduse de utilizator, sînt convertite în valori reale. De exemplu, în cazul programului `P45` utilizatorul poate tasta

```
1 2 3 4<ENTER>
```

Procedura `readln` citeşte datele în acelaşi mod ca şi procedura `read`. Însă, după citirea ultimei valori, restul caracterelor din linia curentă se ignoră. Pentru exemplificare, prezentăm programul `P46`:

```
Program P46;
 { Apelul procedurii readln }
var i,j : integer;
    x,y : real;
begin
  writeln('Apelul procedurii read');
  read(i, j);
  read(x, y);
  writeln('Aţi introdus:');
  writeln('i=', i, ' j=', j, ' x=', x, ' y=', y);
  writeln('Apelul procedurii readln');
```

```
  readln(i, j);
  readln(x, y);
  writeln('Aţi introdus:');
  writeln('i=', i, ' j=', j, ' x=', x, ' y=', y);
end.
```

La execuţia instrucţiunilor

```
read(i, j);
read(x, y)
```

valorile numerice din linia introdusă de utilizator

1 2 3 4<*ENTER*>

vor fi atribuite variabilelor respectiv `i`, `j`, `x`, `y`. La execuţia instrucţiunii

```
readln(i, j)
```

valorile numerice `1` şi `2` din linia

1 2 3 4<*ENTER*>

vor fi atribuite variabilelor `i` şi `j`. Numerele `3` şi `4` se ignoră. În continuare calculatorul execută instrucţiunea

```
readln(x, y)
```

adică va aştepta introducerea unor valori pentru `x` şi `y`.

Subliniem faptul că apelul procedurii `readln` fără parametri va forţa calculatorul să aştepte acţionarea tastei <*ENTER*>. Acest apel se utilizează pentru a suspenda derularea programului, oferindu-i utilizatorului posibilitatea să analizeze rezultatele afişate anterior pe ecran.

Pentru a înlesni introducerea datelor, se recomandă ca apelurile `read(...)` şi `readln(...)` să fie precedate de afişarea unor mesaje sugestive.

Exemple:

```
write('Daţi două numere:'); readln(x, y);
write('Daţi un număr întreg:'); readln(i);
write('x='); readln(x);
write('Răspundeţi D sau N:'); readln(c);
```

Întrebări şi exerciţii

❶ Cum se separă datele numerice ce se introduc de la tastatură?
❷ Care este diferenţa dintre procedurile `read` şi `readln`?
❸ Se consideră următorul program:

```
Program P47;
var i : integer;
    c : char;
    x : real;
begin
```

```
   readln(i);
   readln(c);
   readln(x);
   writeln('i=', i);
   writeln('c=', c);
   writeln('x=', x);
   readln;
end.
```

Precizaţi rezultatele afişate de acest program după tastarea următoarelor date de intrare:

a)	b)	c)	d)
1	1 2 3	123	123 456 789
2	5 6 7	456	abc def ghi
3	8 9 0	789	890 abc def

3.9. INSTRUCŢIUNEA DE EFECT NUL

Executarea acestei instrucţiuni nu are nici un efect asupra variabilelor programului. Sintaxa instrucţiunii în studiu este:

<*Instrucţiunea de efect nul*> ::=

Prin urmare, în textul unui program instrucţiunea de efect nul nu este reprezentată prin nimic. Întrucît instrucţiunile unui program sînt despărţite între ele prin delimitatorul "`;`", prezenţa instrucţiunii de efect nul este marcată de apariţia acestui delimitator.

De exemplu, în textul

```
x:=4;;;; y:=x+1
```

există 5 instrucţiuni dintre care 3 de efect nul.

În mod obişnuit, instrucţiunea de efect nul se utilizează la etapa elaborării şi depănării unor programe complexe. Deşi efectul său la execuţie este nul, inserarea sau eliminarea unei astfel de instrucţiuni (mai exact, a simbolului "`;`") poate să altereze semnificaţia programului.

3.10. INSTRUCŢIUNEA IF

Instrucţiunea de ramificare simplă **`if`**, în funcţie de valoarea unei expresii de tip boolean, decide fluxul execuţiei. Sintaxa instrucţiunii este:

<*Instrucţiune* **`if`**> ::=
  `if` <*Expresie booleană*> **`then`** <*Instrucţiune*>
  [**`else`** <*Instrucţiune*>]

Diagrama sintactică a instrucţiunii în studiu este prezentată în *fig. 3.7.* Expresia booleană din componenţa instrucţiunii **if** se numeşte **condiţie**.

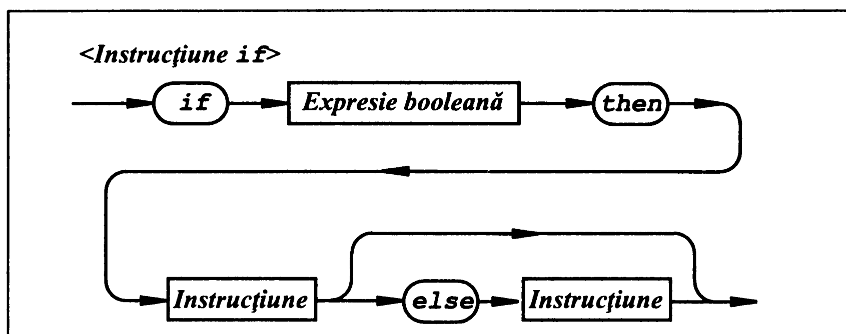


Fig. 3.7. Diagrama sintactică <*Instrucţiune* **if**>

Execuţia instrucţiunii **if** începe prin evaluarea condiţiei. Dacă rezultatul evaluării este `true`, atunci se execută instrucţiunea situată după cuvîntul-cheie **then**. Dacă condiţia are valoarea `false`, atunci: sau se execută instrucţiunea situată după cuvîntul-cheie **else** (dacă există), sau se trece la instrucţiunea situată după instrucţiunea **if**.

În programul ce urmează instrucţiunea **if** este utilizată pentru determinarea maximului a două numere `x` şi `y`, citite de la tastatură.

```
Program P48;
  { Determinarea maximului a două numere }
  var x, y, max : real;
  begin
    writeln('Daţi două numere:');
    write('x='); readln(x);
    write('y='); readln(y);
    if x>=y then max:=x else max:=y;
    writeln('max=', max);
    readln;
  end.
```

Următorul program transformă cifrele romane *I* (unu), *V* (cinci), *X* (zece), *L* (cincizeci), *C* (o sută), *D* (cinci sute) sau *M* (o mie), citite de la tastatură, în numerele corespunzătoare din sistemul zecimal.

```
Program P49;
   {Conversia cifrelor romane }
  var i : integer; c : char;
  begin
    i:=0;
```

```
  writeln('Introduceţi una din cifrele');
  writeln('romane I, V, X, L, C, D, M');
  readln(c);
  if c='I' then i:=1;
  if c='V' then i:=5;
  if c='X' then i:=10;
  if c='L' then i:=50;
  if c='C' then i:=100;
  if c='D' then i:=500;
  if c='M' then i:=1000;
  if i=0  then
  writeln(c, ' - nu este o cifră romană')
  else writeln(i);
  readln;
end.
```

De reţinut că limbajul PASCAL nu consideră simbolul "`;`" ca făcînd parte din instrucţiune, ci îl foloseşte ca delimitator. Prin urmare, dacă într-o instrucţiune

```
if B then S
```

introducem înainte de `S` instrucţiunea cu efect nul

```
if B then; S
```

atunci `S` nu mai intră în componenţa instrucţiunii condiţionale, deci este executată indiferent de valoarea lui `B`.

Dacă într-o instrucţiune

```
if B then I else J
```

introducem după `I` simbolul "`;`", obţinem un program incorect în aspect sintactic:

```
if B then I; else J
```

În acest caz secvenţa **else** `J` este interpretată ca fiind instrucţiunea ce urmează celei condiţionale.

Întrebări şi exerciţii

❶ Care este destinaţia instrucţiunii **if**?

❷ Ce valori va lua variabila `x` după executarea fiecăreia dintre instrucţiunile ce urmează? Se consideră că `a=18`, `b= -15` şi `p=true`.

a) `if a>b then x:=1 else x:=4;`
b) `if a<b then x:=15 else x:=-21;`
c) `if p then x:=32 else x:=638;`
d) `if not p then x:=0 else x:=1;`
e) `if (a<b) and p then x:=-1 else x:=1;`
f) `if (a>b) or p then x:=-6 else x:=-5;`

g) **if** not (a>b) **then** x:=19 **else** x:=-2;
h) **if** (a=b) or p **then** x:=89 **else** x:=-15.

❸ Elaborați un program care calculează valorile uneia dintre funcțiile ce urmează:

a) $y = \begin{cases} 2x, & x \geq 0 ; \\ \frac{x}{2}, & x < 0 ; \end{cases}$ b) $z = \begin{cases} \sin(x+y), & x > 2y ; \\ \cos(x-y), & x \leq 2y ; \end{cases}$

c) $y = \begin{cases} x, & x \geq 3 ; \\ x^2, & x < 3 ; \end{cases}$ d) $y = \begin{cases} x^2, & |x| > 5 ; \\ x^3, & |x| \leq 5 ; \end{cases}$

Exemplu: $y = \begin{cases} 2x^2 + 6, & x > 4 ; \\ x^3 - 3, & x \leq 4 ; \end{cases}$

```
Program P50;
var x,y : real;
begin
  write('x='); readln(x);
  if x>4 then y:=2*sqr(x)+6
  else y:=x*x*x-3;
  writeln('y=', y);
  readln;
end.
```

❹ Ce rezultate va afișa următorul program?

```
Program P51;
var x,y : real;
begin
  write('x='); readln(x);
  y:=x;
  if x>0 then; y:=2*x;
  writeln('y=', y);
  readln;
end.
```

❺ Comentați mesajele afișate pe ecran pe parcursul compilării programului P52:

```
Program P52;
 { Eroare }
var x,y : real;
begin
  write('x='); readln(x);
```

```
  if x>4 then y:=2*sqr(x)+6;
  else y:=x*x*x-3;
  writeln('y=', y); readln;
end.
```

❻ Scrieţi un program care transformă numerele zecimale 1, 5, 10, 50, 100, 500 şi 1000, citite de la tastatură, în cifre romane.

3.11. INSTRUCŢIUNEA CASE

Instrucţiunea de ramificare multiplă **case** conţine o expresie numită **selector** şi o listă de instrucţiuni. Fiecare instrucţiune este prefixată de una sau mai multe constante de caz. Sintaxa instrucţiunii în studiu este:

<*Instrucţiune* **case**> ::=
    case <*expresie*> **of** [<*Caz*> {; <*Caz*>}] [;] **end**
<*Caz*> ::= <*Constantă*> {, <*Constantă*>} : <*Instrucţiune*>

Diagramele sintactice corespunzătoare sînt prezentate în *fig. 3.8.*

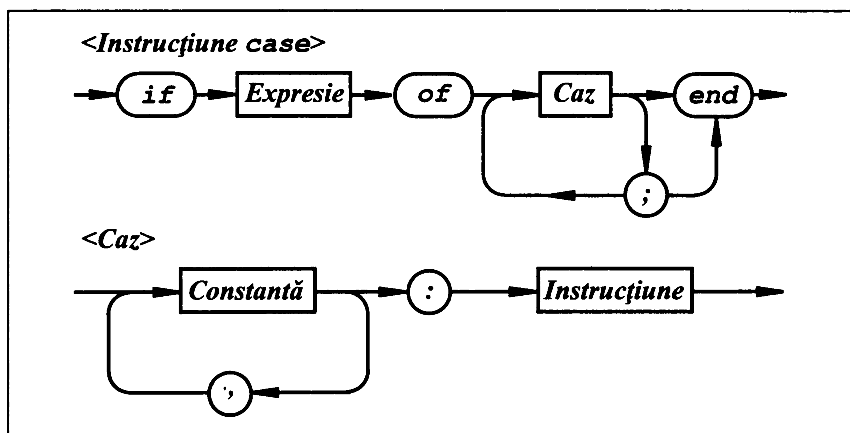


Fig. 3.8. Diagramele sintactice ale instrucţiunii **case**

Selectorul trebuie să fie de tip ordinal. Toate constantele de caz trebuie să fie unice şi compatibile din punct de vedere al atribuirii cu tipul selectorului.

Exemple:

```
var i : integer; c : char; a, b, y : real;
    case i of
      0, 2, 4, 6, 8 : writeln('Cifră pară');
      1, 3, 5, 7, 9 : writeln('Cifră impară');
```

```
end;
case c of
  '+' : y:=a+b;
  '-' : y:=a-b;
  '*' : y:=a*b;
  '/' : y:=a/b;
end;
```

Execuţia instrucţiunii **case** începe prin evaluarea selectorului. În funcţie de valoarea obţinută, se execută instrucţiunea prefixată de constanta respectivă.

În programul ce urmează instrucţiunea **case** este utilizată pentru conversia cifrelor romane în numere zecimale.

```
Program P53;
  { Conversia cifrelor romane }
  var i : integer; c : char;
  begin
    i:=0;
    writeln('Introduceţi una din cifrele');
    writeln('romane I, V, X, L, C, D, M');
    readln(c);
  case c of
      'I' : i:=1;
      'V' : i:=5;
      'X' : i:=10;
      'L' : i:=50;
      'C' : i:=100;
      'D' : i:=500;
      'M' : i:=1000;
  end;
  if i=0 then
    writeln(c, ' - nu este o cifră romană')
  else writeln(i);
  readln;
  end.
```

Subliniem faptul că în unele implementări ale limbajului sintaxa şi semantica instrucţiunii **case** au fost modificate. Lista cazurilor poate să includă o instrucţiune precedată de cuvîntul-cheie **else** (în unele versiuni **otherwise**). Constantele de caz pot fi înlocuite cu intervale de forma:

<Constantă> . . *<Constantă>*

Exemplu (Turbo Pascal 7.0):

```
Program P54;
  { Simularea unui calculator de buzunar }
  var a,b : real; c : char;
  begin
    write('a='); readln(a);
```

```
    write('b='); readln(b);
    write('Cod operaţie '); readln(c);
    case c of
      '+' : writeln('a+b=', a+b);
      '-' : writeln('a-b=', a-b);
      '*' : writeln('a*b=', a*b);
      '/' : writeln('a/b=', a/b);
    else writeln('Cod operaţie necunoscut');
    end;
    readln;
  end.
```

Întrebări şi exerciţii

❶ Indicaţi pe diagramele sintactice din *fig. 3.8* drumurile care corespund instrucţiunilor **case** din programele P53 şi P54.

❷ Cum se execută o instrucţiune **case**? De ce tip trebuie să fie selectorul?

❸ Ce fel de constante pot fi utilizate ca constante de caz?

❹ Înlocuiţi instrucţiunea **case** a programului P54 cu o secvenţă echivalentă de instrucţiuni **if**.

❺ Utilizînd instrucţiunea **case**, scrieţi un program care transformă numerele zecimale 1, 5, 10, 50, 100, 500, 1000, citite de la tastatură, în cifre romane.

❻ Ce va apare pe ecran în urma execuţiei programului P55?

```
Program P55;
type Semnal=(Rosu, Galben, Verde);
var s : Semnal;
begin
    s:=Verde;
    s:=pred(s);
    case s of
      Rosu   : writeln('STOP');
      Galben : writeln('ATENŢIE');
      Verde  : writeln('START');
    end;
    readln;
end.
```

❼ Comentaţi programele:

```
Program P56;
 { Eroare }
var x : real;
begin
  writeln('x='); readln(x);
  case x of
    0,2,4,6,8 : writeln('Cifră pară');
    1,3,5,7,9 : writeln('Cifră impară');
  end;
  readln;
end.
```

```
Program P57;
 { Eroare }
var i : 1..4;
begin
  write('i='); readln(i);
  case i of
    1 : writeln('unu');
    2 : writeln('doi');
    3 : writeln('trei');
    4 : writeln('patru');
    5 : writeln('cinci');
  end;
  readln;
end.
```

```
Program P58;
 { Eroare }
type  Semnal=(Rosu, Galben, Verde);
  Culoare=(Albastru, Portocaliu);
var s : Semnal;
  c : Culoare;
begin
  { ... }
  case s of
    Rosu : writeln('STOP');
    Galben : Writeln('ATENŢIE');
    Verde : writeln('START');
    Albastru : writeln('PAUZĂ');
  end;
  { ... }
end.
```

3.12. INSTRUCŢIUNEA FOR

Instrucţiunea **for** indică execuţia repetată a unei instrucţiuni în funcţie de valoarea unei variabile de control. Sintaxa instrucţiunii în studiu este

<*Instrucţiune* for> ::=
  for <*Variabilă*> ::= <*Expresie*> <*Pas*> <*Expresie*>
  do <*Instrucţiune*>

<*Pas*> ::= **to** | **downto**

Diagramele sintactice sînt prezentate în *fig. 3.9*.

Variabila situată după cuvîntul-cheie **for** se numeşte **variabilă de control** sau **contor**. Această variabilă trebuie să fie de tip ordinal.

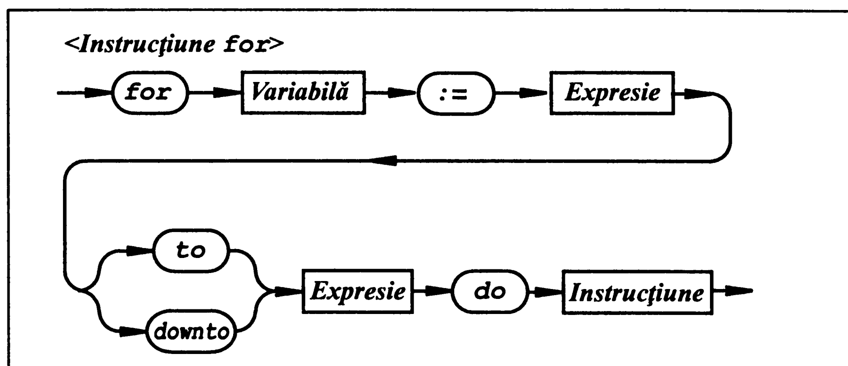


Fig. 3.9. Diagrama sintactică a instrucţiunii **for**

Valorile expresiilor din componenţa instrucţiunii **for** trebuie să fie compatibile, în aspectul atribuirii, cu tipul variabilei de control. Aceste expresii sînt evaluate o singură dată, la începutul ciclului. Prima expresie indică valoarea iniţială, iar expresia a doua – valoarea finală a variabilei de control.

Instrucţiunea situată după cuvîntul-cheie **do** se execută pentru fiecare valoare din domeniul determinat de valoarea iniţială şi de valoarea finală.

Dacă instrucţiunea **for** utilizează pasul **to**, valorile variabilei de control sînt incrementate la fiecare repetiţie, adică se trece la succesorul valorii curente. Dacă valoarea iniţială este mai mare decît valoarea finală, instrucţiunea situată după cuvîntul-cheie **do** nu se execută nici o dată.

Dacă instrucţiunea **for** utilizează pasul **downto**, valorile variabilei de control sînt decrementate la fiecare repetiţie, adică se trece la predecesorul valorii curente. Dacă valoarea iniţială este mai mică decît valoarea finală, instrucţiunea situată după cuvîntul-cheie **do** nu se execută nici o dată.

Exemplu :

```
Program P59;
  { Instrucţiunea for }
  var i : integer; c : char;
  begin
  for i:=0 to 9 do write(i:2);
    writeln;
  for i:=9 downto 0 do write(i:2);
    writeln;
  for c:='A' to 'Z' do write(c:2);
    writeln;
  for c:='Z' downto 'A' do write(c:2);
    writeln;
```

```
   readln;
end.
```

Rezultatele afişate pe ecran:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Valorile variabilei de control nu pot fi modificate în interiorul ciclului, adică :

1) nu se fac atribuiri variabilei de control;

2) variabila actuală de control nu poate fi variabilă de control a altei instrucţiuni **for** incluse;

3) nu se admit apeluri de tipul `read`, `readln` în care apare variabila de control.

La ieşirea din instrucţiunea **for** valoarea variabilei de control nu este definită, în afara cazului cînd ieşirea din ciclu se face forţat, printr-o instrucţiune de salt necondiţionat **goto**.

Instrucţiunea **for** este utilă pentru programarea algoritmilor iterativi, în care numărul de repetări este cunoscut. Pentru exemplificare prezentăm programele `P60`, `P61` şi `P62` care calculează respectiv $n!$, x^n *şi suma* $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}$.

```
Program P60;
 { Calcularea factorialului }
var n, i, f : 0..MaxInt;
begin
  write('n='); readln(n);
  f:=1;
  for i:=1 to n do f:=f*i;
  writeln('n!=', f);
  readln;
end.
```

```
Program P61;
  { Calcularea lui x la puterea n }
  var x, y : real;
    n, i : 0..MaxInt;
  begin
    write('x='); readln(x);
    write('n='); readln(n);
    y:=1;
    for i:=1 to n do y:=y*x;
    writeln('y=', y);
    readln;
  end.
```

```
Program P62;
  { Calcularea sumei 1 + 1/2 + 1/3 + ... + 1/n }
  var n, i : 1..MaxInt;
    s : real;
  begin
    write('n=');
    readln(n);
    s:=0;
   for i:=1 to n do s:=s+1/i;
    writeln('s=', s);
   readln;
  end.
```

Întrebări şi exerciţii

❶ Indicaţi pe diagrama sintactică din *fig. 3.9* drumurile care corespund instrucţiunilor **for** din programul P59.

❷ Cum se execută o instrucţiune **for**?

❸ Ce va afişa pe ecran programul P63?

```
Program P63;
  type Zi=(L, Ma, Mi, J, V, S, D);
  var z : Zi;
  begin
    for z:=L to S do writeln(ord(z));
      readln;
    for z:=D downto Ma do writeln(ord(z));
      readln;
  end.
```

❹ Se consideră declaraţiile :

```
var i, j, n : integer;
    x, y : real;
    c : char;
```

Care din instrucţiunile ce urmează sînt corecte?

a) `for i:=-5 to 5 do j:=i+3;`
b) `for i:=-5 to 5 do i:=j+3;`
c) `for j:=-5 to 5 do i:=j+3;`
d) `for i:=1 to n do y:=y/i;`
e) `for x:=1 to n do y:=y/x;`
f) `for c:='A' to 'Z' do writeln(ord(c));`
g) `for c:='Z' downto 'A' do writeln(ord(c));`
h) `for i:=-5 downto -10 do readln(i);`
i) `for i:=ord('A') to ord('A')+9 do writeln(i);`
j) `for c:='0' to '9' do writeln(c,ord(c):3);`
k) `for j:=i/2 to i/2+10 do writeln(j);`

❺ Se consideră declaraţiile:

```
var i, m, n : integer;
```

De cîte ori se vor executa apelurile writeln(i) şi writeln(2*i) din componenţa instrucţiunilor

```
for i:=m to n do writeln(i);
for i:=m to n do writeln(2*i);
```

dacă:

a) m=1, n=5;
b) m=3, n=5;
c) m=3, n=3;
d) m=5, n=3?

❻ Elaboraţi un program care afişează pe ecran codurile caracterelor *'A'*, *'B'*, *'C'*, . . . , *'Z'*.

❼ Calculaţi pentru primii *n* termeni:

a) $1+3+5+7+...$ şi $1 \cdot 3 \cdot 5 \cdot 7 \cdot ...$;
b) $2+4+6+8+...$ şi $2 \cdot 4 \cdot 6 \cdot 8 \cdot ...$;
c) $3+6+9+12+...$ şi $3 \cdot 6 \cdot 9 \cdot 12 \cdot ...$;
d) $4+8+12+16+...$ şi $4 \cdot 8 \cdot 12 \cdot 16 \cdot ...$;

Exemplu: Pentru *n*=3 avem $1+3+5=9$; $1 \cdot 3 \cdot 5=15$.

❽ Calculaţi suma primilor *n* termeni:

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\frac{1}{5}-\frac{1}{6}...$$

Indicaţie: Utilizaţi în ciclu instrucţiunea `if odd(...) then ... else ...` .

3.13. INSTRUCŢIUNEA COMPUSĂ

Instrucţiunea în studiu are următoarea sintaxă:

<*Instrucţiune compusă*> ::=
    begin <*Instrucţiune*> {; <*Instrucţiune*>} **end**

Diagrama sintactică este prezentată în *fig. 3.10.*

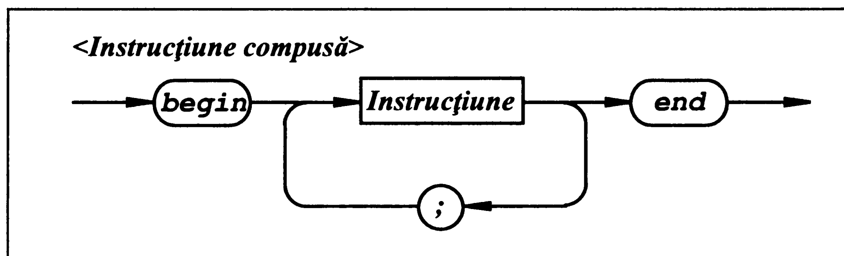


Fig. 3.10. Diagrama sintactică <*Instrucţiune compusă*>

Exemple:

```
1) begin
     a:=x+12;
     p:=q and r;
     writeln(p)
   end;
2) begin
     write('x=');
     readln(x)
   end;
```

Cuvintele-cheie **begin** şi **end** joacă rolul unor "paranteze". Mulţimea instrucţiunilor cuprinse între aceste paranteze, din punctul de vedere al limbajului, formează o singură instrucţiune. Prin urmare, instrucţiunea compusă este utilă, pentru a plasa mai multe instrucţiuni în locurile din programe în care este permisă numai o singură instrucţiune (vezi instrucţiunile **if**, **case**, **for** ş.a.).

Exemple:

```
1) if a>0 then begin
     x:=a+b;
     y:=a*b
   end;
   else begin
     x:=a-b;
     y:=a/b
   end;
2) case c of
   '+' : begin y:=a+b; writeln('Adunarea') end;
   '-' : begin y:=a-b; writeln('Scăderea') end;
   '*' : begin y:=a*b; writeln('Înmulţirea') end;
   '/' : begin y:=a/b; writeln('Împărţirea') end;
   end ;
3) for i:=1 to n do
   begin
     write('x=');
     readln(x);
     s:=s+x
   end;
```

De menţionat că partea executabilă a oricărui program este o instrucţiune compusă, adică o secvenţă de instrucţiuni încadrată între "parantezele" **begin** şi **end**.

Întrucît simbolul "`;`" nu termină, ci separă instrucţiunile, el nu este necesar înaintea cuvîntului-cheie **end**. Cu toate acestea, mulţi programatori inserează acest simbol cu scopul de a continua, în caz de necesitate, lista respectivă de instrucţiuni. Amintim că apariţia adiţională a unui simbol "`;`" semnifică inserarea unei instrucţiuni de efect nul.

Pentru a face programele mai lizibile, cuvintele **begin** şi **end** se scriu strict unul sub altul, iar instrucţiunile dintre "paranteze" — cu cîteva poziţii mai la dreapta. Dacă instrucţiunea compusă **begin**...**end** este inclusă în componenţa altor instrucţiuni (**if**, **case**, **for** ş.a.), cuvintele-cheie **begin** şi **end** se scriu deplasate la dreapta.

Pentru exemplificare prezentăm programul P64 în care se calculează media aritmetică a *n* numere, citite de la tastatură.

```
Program P64;
  { Media aritmetică a n numere }
  var x, suma, media : real; i, n : integer;
  begin
    write('n='); readln(n);
    suma:=0;
    writeln('Daţi ', n, ' numere:');
    for i:=1 to n do
      begin
        write('x='); readln(x);
        suma:=suma+x;
      end;
    if n>0 then
          begin
            media:=suma/n;
            writeln('media=', media);
          end
      else writeln('media=*****');
      readln;
  end.
```

Întrebări şi exerciţii

❶ Care este destinaţia instrucţiunii compuse?

❷ Indicaţi pe diagrama sintactică din *fig. 3.10* drumurile care corespund instrucţiunii compuse din programul P64.

❸ Elaboraţi un program care citeşte de la tastatură *n* numere şi afişează pe ecran:

a) suma şi media aritmetică a numerelor citite;
b) suma şi media aritmetică a numerelor pozitive;
c) suma şi media aritmetică a numerelor negative.

❹ Elaboraţi un program care citeşte de la tastatură *n* caractere şi afişează pe ecran:

a) numărul cifrelor zecimale citite;
b) numărul cifrelor pare;
c) numărul cifrelor impare;
d) numărul literelor citite;
e) numărul vocalelor;
f) numărul consoanelor.

Caracterele introduse se separă prin acționarea tastei <*ENTER*>. Sînt admise cifrele zecimale 0, 1, 2, ..., 9 şi literele mari *A, B, C, ..., Z* ale alfabetului latin.

3.14. INSTRUCŢIUNEA `WHILE`

Instrucţiunea **`while`** conţine o expresie booleană care controlează execuţia repetată a altei instrucţiuni. Sintaxa instrucţiunii în studiu este:

<*Instrucţiune* `while`> ::=
  `while` <*Expresie booleană*> **do** <*Instrucţiune*>

Diagrama sintactică este prezentată în *fig. 3.11*.

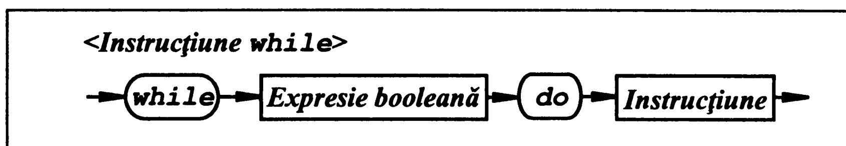


Fig. 3.11. Diagrama sintactică a instrucţiunii **`while`**

Exemple:

```
1) while x>0 do x:=x-1;
2) while x<3.14 do
   begin
     x:=x+0.001;
     writeln(sin(x));
   end;
3) while p do
   begin
     x:=x+0.001;
     y:=10*x;
     p:=y<1000;
   end;
```

Instrucţiunea situată după cuvîntul-cheie **do** se execută repetat atîta timp, cît valoarea expresiei booleene este `true`. Dacă expresia booleană ia valoarea `false`, instrucţiunea de după **do** nu se mai execută. Se recomandă ca expresia booleană să fie cît mai simplă, deoarece ea este evaluată la fiecare iteraţie.

În mod obişnuit, instrucţiunea **`while`** se utilizează pentru organizarea calculelor repetitive cu variabile de control de tip `real`.

În programul ce urmează, instrucţiunea **`while`** este utilizată pentru afişarea valorilor funcţiei $y = 2x$. Argumentul x ia valori de la x_1 la x_2 cu pasul Δx.

```
Program P65;
 { Tabelul funcţiei y=2*x }
  var x, y,x1, x2, deltaX : real;
  begin
    write('x1='); readln(x1);
    write('x2='); readln(x2);
    write('deltaX='); readln(deltaX);
    writeln('x':10, 'y':20);
    writeln;
    x:=x1;
    while x<=x2 do
    begin
      y:=2*x;
      writeln(x:20, y:20);
      x:=x+deltaX;
    end;
    readln;
  end.
```

Instrucţiunea **while** este deosebit de utilă în situaţia în care numărul de execuţii repetate ale unei secvenţe de instrucţiuni e dificil de evaluat.

Pentru exemplificare prezentăm programul P66, care afişează pe ecran media aritmetică a numerelor pozitive citite de la tastatură.

```
Program P66;
 {Media aritmetică a numerelor pozitive
  citite de la tastatură }
  var x, suma : real; n : integer;
  begin
    n:=0;
    suma:=0;
    writeln('Daţi numere pozitive:');
    readln(x);
    while x>0 do
    begin
      n:=n+1;
      suma:=suma+x;
      readln(x);
    end;
    writeln('Aţi introdus ', n, ' numere pozitive.');
  if n>0 then writeln('media=', suma/n)
    else writeln('media=*****');
    readln;
end.
```

Se observă că numărul de execuţii repetate ale instrucţiunii compuse **begin** ... **end** din componenţa instrucţiunii **while** nu poate fi calculat din timp. Execuţia instrucţiunii **while** se termină cînd utilizatorul introduce un număr $x \leq 0$.

Întrebări şi exerciţii

❶ Cum se execută o instrucţiune **while**?

❷ Indicaţi pe diagramele sintactice din *fig. 3.11* drumurile care corespund instrucţiunilor **while** din programele P65 şi P66.

❸ Utilizînd instrucţiunea **while**, scrieţi un program care afişează pe ecran valorile funcţiei $y = f(x)$ pentru valori ale argumentului de la x_1 la x_2 cu pasul Δx:

a) $y = x^2$;

b) $y = \begin{cases} \sin x, & x \geq 0 ; \\ \cos x, & x < 0 ; \end{cases}$

c) $y = \begin{cases} \sqrt{x}, & x \geq 0 ; \\ x^2, & x < 0 ; \end{cases}$

d) $y = \begin{cases} x^2, & |x| > 5 ; \\ x^3, & |x| \leq 5 ; \end{cases}$

❹ Utilizatorul introduce de la tastatură numere întregi pozitive, separate prin acţionarea tastei <*ENTER*>. Sfîrşitul secvenţei de numere e indicat prin introducerea numărului 0. Scrieţi un program care afişează pe ecran:

a) suma şi media aritmetică a numerelor pare;
b) suma şi media aritmetică a numerelor impare.

❺ Scrieţi un program care afişează pe ecran valorile funcţiei $z = f(x,y)$. Argumentul x ia valori de la x_1 la x_2 cu pasul Δx; argumentul y ia valori de la y_1 la y_2 cu pasul Δy:

a) $z = \begin{cases} x+y, & x \geq y ; \\ x-y, & x < y ; \end{cases}$

b) $z = \begin{cases} \sin(x+y), & x > 2y ; \\ \cos(x+y), & x \leq 2y ; \end{cases}$

c) $z = \begin{cases} x^2+y^2, & x+y > 6 ; \\ x^3-y^3, & x+y \leq 6 ; \end{cases}$

d) $z = \begin{cases} x+y, & x > 8y ; \\ \text{abs}\,(x+y), & x \leq 8y ; \end{cases}$

Exemplu: $z = x + y$.

```
Program P67;
 { Tabelul funcţiei z=x+y }
var x, y, z,
    x1, x2, deltaX,
    y1, y2, deltaY : real;
begin
    write('x1=');
    readln(x1);
    write('x2=');
    readln(x2);
    write('deltaX=');
    readln(deltaX);
    write('y1=');
    readln(y1);
    write('y2=');
    readln(y2);
    write('deltaY=');
    readln(deltaY);
```

```
    writeln('x':10, 'y':20, 'z':20);
    writeln;
    x:=x1;
  while x<=x2 do
  begin
      y:=y1;
      while y<=y2 do
      begin
        z:=x+y;
        writeln(x:20, y:20, z:20);
        y:=y+deltaY;
      end;
      x:=x+deltaX;
  end;
  readln;
end.
```

❻ Instrucţiunea repetitivă

```
for i:=i1 to i2 do writeln(ord(i));
```

este echivalentă cu secvenţa de instrucţiuni

```
i:=i1;
while i<=i2 do
begin
  writeln(ord(i));
  i:=succ(i);
end.
```

Scrieţi o secvenţă echivalentă pentru instrucţiunea repetitivă

```
for i:=i1 downto i2 do writeln(ord(i)).
```

❼ Se consideră declaraţiile

```
var x1, x2, deltaX : real;
i, n : integer;
```

Care din secvenţele de instrucţiuni ce urmează sînt echivalente?

```
a)  x:=x1;
    while x<=x2 do
    begin
      writeln(x);
      x:=x+deltaX;
    end;
b)  n:=trunc((x2-x1)/deltaX)+1;
    x:=x1;
    for i:=1 to n do
    begin
      writeln(x);
      x:=x+deltaX;
    end;
```

```
c)  n:=round((x2-x1)/deltaX)+1;
    x:=x1;
    for i:=1 to n do
    begin
      writeln(x);
      x:=x+deltaX;
    end;
```

Argumentaţi răspunsul.

3.15. INSTRUCŢIUNEA REPEAT

Instrucţiunea **repeat** indică repetarea unei secvenţe de instrucţiuni în funcţie de valoarea unei expresii booleene.

Sintaxa instrucţiunii este:

<*Instrucţiune* repeat> ::= **repeat** <*Instrucţiune*>
{;<*Instrucţiune*>} **until** <*Expresie booleană*>

Diagrama sintactică este prezentată în *fig. 3.12*.

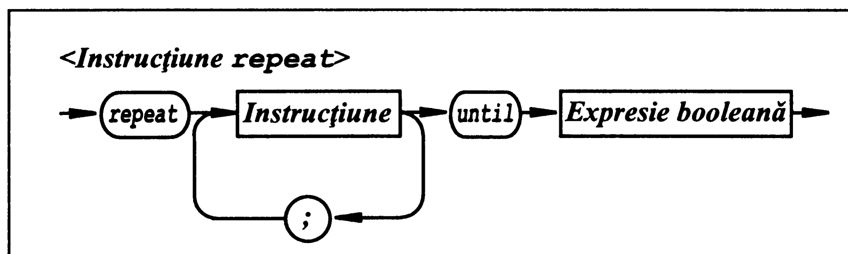


Fig. 3.12. Diagrama sintactică a instrucţiunii **repeat**

Exemple:

```
1)  repeat x:=x-1 until x<0;
2)  repeat
      y:=y+delta;
      writeln(y)
    until y>20.5;
3)  repeat
      readln(i);
      writeln(odd(i))
    until i=0;
```

Instrucţiunile situate între cuvintele-cheie **repeat** şi **until** se execută repetat atît timp, cît expresia booleeană este falsă. Cînd această expresie devine adevărată, se trece la instrucţiunea următoare. Evident, instrucţiunile aflate între **repeat** şi **until** vor fi executate

cel puţin o dată, deoarece evaluarea expresiei logice are loc după ce s-a executat această secvenţă.

În mod obişnuit, instrucţiunea **repeat** se utilizează în locul instrucţiunii while atunci cînd evaluarea expresiei care controlează repetiţia se face după executarea secvenţei de repetat.

Programul ce urmează afişează la ecran paritatea numerelor întregi citite de la tastatură.

```
Program P68;
 { Paritatea numerelor citite de la tastatură }
var i : integer;
begin
  writeln('Daţi numere întregi:');
  repeat
    readln(i);
    if odd(i) then writeln(i:6, ' - număr impar')
    else writeln(i:6, ' - număr par');
  until i=0;
readln;
end.
```

Execuţia instrucţiunii **repeat** se termină cînd utilizatorul introduce i=0.

Următorul program calculează valoarea $y = \sqrt{x}$, $x > 0$, utilizînd relaţia recurentă:

$$y_i = \frac{y_{i-1}}{2} + \frac{x}{2y_{i-1}}, \quad y_1 = 1.$$

Iteraţiile se termină cînd pentru aproximările succesive y_{i-1} şi y_i se respectă condiţia

$$|y_{i-1} - y_i| \leq \varepsilon.$$

```
Program P69;
{ Calcularea rădăcinii pătrate }
  var x,epsilon,y1,y2:real;
  begin
  write('x='); readln(x);
  write('epsilon='); readln(epsilon);
  y2:=1;
  repeat
    y1:=y2; y2:=y1/2+x/(2*y1);
    writeln(y2);
  until abs(y1-y2)<=epsilon;
  readln;
end.
```

Parametrul ε specifică precizia cu care se va calcula $\sqrt{x}$. De exemplu, pentru $x = 2$ şi $\varepsilon = 0.01$ programul `P69` va afişa:

```
1.5000000000E+00
1.4166666667E+00
1.4142156863E+00,
```

iar pentru $x = 2$ şi $\varepsilon = 0.0000000001$:

```
1.5000000000E+00
1.4166666667E+00
1.4142156863E+00
1.4142135624E+00
1.4142135624E+00.
```

Din exemplele studiate se observă că instrucţiunea `repeat` este utilă în situaţia în care numărul de executări repetate ale unei secvenţe de instrucţiuni este dificil de evaluat.

Întrebări şi exerciţii

❶ Cum se execută o instrucţiune **repeat**?

❷ Indicaţi pe diagramele sintactice din *fig. 3.12* drumurile care corespund instrucţiunilor **repeat** din programele `P68` şi `P69`.

❸ Se consideră instrucţiunile:

a) **repeat**
<*Instrucţiune 1*>
<*Instrucţiune 2*>
. . .
<*Instrucţiune n*>
until p;

b) **while not** p **do**
begin
<*Instrucţiune 1*>
<*Instrucţiune 2*>
. . .
<*Instrucţiune n*>
end.

Sînt oare echivalente aceste instrucţiuni? Argumentaţi răspunsul.

❹ Elaboraţi un program care citeşte de la tastatură o secvenţă de caractere şi afişează pe ecran:

a) numărul cifrelor zecimale citite;
b) numărul cifrelor pare;
c) numărul cifrelor impare.

Caracterele introduse se separă prin acţionarea tastei <*ENTER*>. Sînt admise cifrele zecimale 0, 1, 2, ..., 9 şi caracterul * care indică sfîrşitul secvenţei.

❺ Elaboraţi un program pentru calcularea valorii $y = \sqrt[3]{x}$ cu ajutorul relaţiei recurente:

$$y_i = \frac{2y_{i-1}}{3} + \frac{x}{3y_{i-1}^2}, \quad y_1 = 1.$$

❻ Este oare echivalentă instrucţiunea

```
for i:=i1 to i2 do writeln(ord(i))
```

cu secvenţa de instrucţiuni

```
i:=i1;
repeat
  writeln(ord(i));
  i:=succ(i)
until i>i2;
```

Argumentaţi răspunsul.

❼ Scrieţi un program care afişează pe ecran valorile funcţiei $y = f(x)$. Argumentul x ia valori de la x_1 la x_2 cu pasul Δx. Ciclul se va organiza cu ajutorul instrucţiunii **repeat**.

a) $y = x^4$;

b) $y = \sqrt[4]{x}$;

c) $y = \begin{cases} 3x, & x \geq 4 \, ; \\ \frac{x}{3}, & x < 4 \, ; \end{cases}$

d) $y = \begin{cases} 16 - \frac{x}{3}, & x > 0 \, ; \\ 26 + x^2, & x \leq 0. \end{cases}$

❽ Elaboraţi un program care citeşte de pe tastatură o secvenţă de caractere şi afişează pe ecran:

a) numărul literelor citite;
b) numărul literelor mari;
c) numărul literelor mici.

Caracterele introduse se separă prin acţionarea tastei <*ENTER*>. Sînt admise literele mari şi mici ale alfabetului latin şi caracterul * care indică sfîrşitul secvenţei.

3.16. INSTRUCŢIUNEA GOTO

Instrucţiunile unui program sînt executate secvenţial, aşa cum apar scrise în textul programului. Instrucţiunea de salt necondiţionat **goto** oferă posibilitatea de a întrerupe această secvenţă şi de a relua execuţia dintr-un alt loc al programului. Instrucţiunea în studiu are sintaxa:

<*Instrucţiune goto*> ::= **goto** <*Etichetă*>

Amintim că eticheta este un număr întreg fără semn care prefixează o instrucţiune a programului (*fig. 3.1*). Etichetele unui program sînt listate în partea declarativă a programului după cuvîntul-cheie **label**. Sintaxa acestei declaraţii este:

<*Etichete*> ::= **label** <*Etichetă*> {, <*Etichetă*>};

Diagramele sintactice ale unităţilor gramaticale în studiu sînt prezentate în *fig. 3.13*. De reţinut că declararea prin **label** a etichetelor este obligatorie.

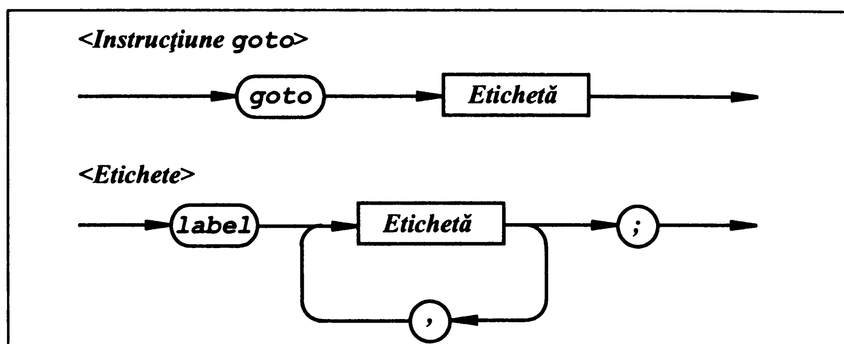


Fig. 3.13. Diagramele sintactice <*Instrucţiune* **goto**> şi <*Etichete*>

Execuţia instrucţiunii **goto** are ca efect transferul controlului la instrucţiunea prefixată de eticheta respectivă.

Pentru exemplificare prezentăm programul P70 care calculează valoarea funcţiei

$$y = \begin{cases} x, & x \geq 0; \\ 2x, & x < 0. \end{cases}$$

Valorile argumentului *x* se citesc de la tastatură.

```
Program P70;
 {Execuţia instrucţiunii goto }
label 1, 2;
var x, y : real;
begin
  write('x='); readln(x);
  if x>=0 then goto 1;
  y:=2*x;
  writeln('x<0, y=', y);
  goto 2;
1 : y:=x;
  writeln('x>=0, y=', y);
2 : readln;
end.
```

Dacă utilizatorul tastează o valoare $x \geq 0$, se va executa instrucţiunea **goto** 1 şi controlul va trece la instrucţiunea de atribuire

`1: y:=x`

După aceasta se vor executa instrucţiunile

```
    writeln('x>=0 , y=', y);
  2: readln
```

Dacă utilizatorul tastează o valoare $x<0$, se execută instrucţiunile

```
    y:= 2*x;
    writeln('x<0 , y=', y);
    goto 2
```

Ultima instrucţiune va transfera controlul instrucţiunii

```
  2: readln
```

Etichetele şi instrucţiunile unui program trebuie să respecte următoarele reguli:

1) orice etichetă trebuie să fie declarată cu ajutorul cuvîntului-cheie **label**;

2) orice etichetă trebuie să prefixeze o singură instrucţiune;

3) este interzis saltul din afara unei instrucţiuni structurate (**if**, **for**, **while**, ... ş.a.) în interiorul ei.

Exemple:

```
  if i>5 then 1: writeln('i>5')
  else writeln('i<=5');
  ...
  goto 1; {Eroare}

  for i:=1 to 10 do
  begin
10: writeln('i=', i);
  end;
  ...
  goto 10; {Eroare}
```

În lipsa lui **goto** instrucţiunile unui program sînt executate în ordinea în care sînt scrise. Prin urmare, instrucţiunile **goto** violează concordanţa dintre textul programului şi ordinea de excuţie a instrucţiunilor. Acest fapt complică elaborarea, verificarea şi depanarea programelor. În consecinţă, folosirea instrucţiunii **goto** nu este recomandată.

De exemplu, programul P70 poate fi refăcut după cum urmează:

```
Program P71;
 { Excluderea instrucţiunii goto din programul P70 }
var x, y : real;
begin
  write('x=');
  readln(x);
  if x>=0 then
  begin
    y:=x;
    writeln('x>=0, y=', y);
```

```
  end
  else
  begin
    y:=2*x;
    writeln('x<0, y=', y);
  end;
  readln;
end.
```

De regulă, instrucţiunea **goto** se utilizează în cazuri extraordinare, de exemplu, pentru a mări viteza de derulare sau pentru a micşora lungimea unui program.

Întrebări şi exerciţii

❶ Care este destinaţia instrucţiunii **goto**?

❷ Indicaţi pe diagramele sintactice din *fig. 3.13* drumurile care corespund etichetelor şi instrucţiunilor **goto** din programul P70.

❸ Transcrieţi fără a utiliza instrucţiunea **goto**:

```
Program P72;
 { Afişarea formulelor de salut }
label 1, 2, 3;
var i : 6..23;
begin
  write('Cît e ora?'); readln(i);
  if i>12 then goto 1;
  writeln('Bună dimineaţa!');
  goto 3;
1:  if i>17 then goto 2;
  writeln('Bună ziua!');
  goto 3;
2:  writeln('Bună seara!');
3:  readln;
end.
```

❹ Comentaţi următorul program:

```
Program P73;
 { Eroare }
label 1;
var i : 1..5;
begin
  i:=1;
1:  writeln(i);
  i:=i+1;
  goto 1;
end.
```

❺ Ce va afişa pe ecran programul ce urmează?

```
Program P74;
 { Eroare }
label 1;
var x: real;
begin
  x:=0;
1:  writeln(x);
  x:=x+1e-30;
  goto 1;
end.
```

Amintim că derularea unui program poate fi întreruptă prin acționarea tastelor *<CTRL+C>* sau *<CTRL+BREAK>*.

❻ Comentați programul ce urmează :

```
Program P75;
 { Eroare }
label 1;
var i : integer;
begin
  i:=1;
  while i<=20 do
  begin
    writeln(i);
1:  i:=i+1;
  end; goto 1;
end.
```

3.17. GENERALITĂŢI DESPRE STRUCTURA UNUI PROGRAM PASCAL

Un program PASCAL are următoarea structură:

<Program> ::=
  <Antet program>
  <Corp> .

Antetul specifică numele programului şi include, opţional, o listă de parametri formali:

<Antet program> ::=
  Program *<Identificator>*
    [(*<Identificator>* {, *<Identificator>*})];

Exemple:

```
Program A1;
Program B6(Intrare);
Program C15(Intrare, Iesire);
```

În mod obişnuit, parametrii formali se folosesc pentru comunicarea programului cu mediul său. Aceştia vor fi studiaţi mai amănunţit în capitolele următoare.

Corpul unui program este compus din partea declarativă şi partea executabilă:

<*Corp*> ::= <*Declaraţii*>
<*Instrucţiune compusă*>

Partea declarativă a programului are următoarea sintaxă:

<*Declaraţii*> ::= [<*Etichete*>]
[<*Constante*>]
[<*Tipuri*>]
[<*Variabile*>]
[<*Subprograme*>]

Subprogramele vor fi studiate în capitolele următoare.

Partea executabilă a unui program este o instrucţiune compusă **begin**...**end**.

Diagramele sintactice ale unităţilor gramaticale în studiu sînt prezentate în *fig. 3.14*.

Subliniem faptul că sfîrşitul unui program PASCAL este indicat de simbolul **"." (punct).**

Pentru exemplificare prezentăm în continuare programul P76 care calculează lungimea arcului de cerc de α grade şi aria sectorului respectiv.

```
Program P76;
  { Lungimea arcului de cerc
  şi aria sectorului respectiv }
label 1, 2;
const Pi=3.141592654;
type grade=0..360;
var alfa : grade;
    raza, lungimea, aria : real;
begin
  write('raza='); readln(raza);
  if raza<0 then goto 1;
  write('alfa='); readln(alfa);
  lungimea:=Pi*raza*alfa/180;
  writeln('lungimea=', lungimea);
  aria:=Pi*sqr(raza)*alfa/360;
  writeln('aria=', aria);
  goto 2;
1:  writeln('Eroare: raza<0');
2:  readln;
end.
```

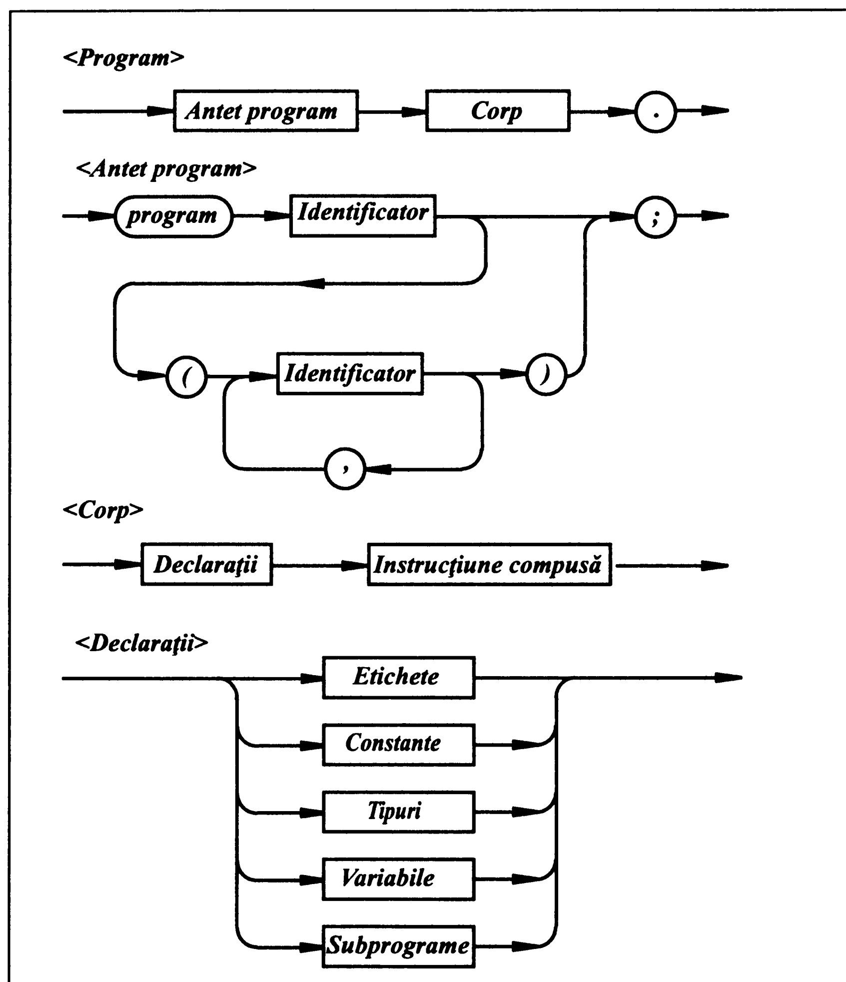


Fig. 3.14. Structura unui program PASCAL

Întrebări şi exerciţii

❶ Care este destinaţia antetului de program? Cum se indică sfîrşitul unui program PASCAL?

❷ Indicaţi pe diagramele sintactice din *fig. 3.14* drumurile care corespund unităţilor gramaticale ale programului `P76`.

❸ Transcrieţi programul `P76` fără a utiliza instrucţiunea **goto**. Indicaţi partea declarativă şi partea executabilă a programului elaborat.

❹ Care este destinaţia parametrilor dintr-un antet de program?

Capitolul 4

TIPURI DE DATE STRUCTURATE

4.1. TIPURI DE DATE *TABLOU* (ARRAY)

Mulţimea de valori ale unui tip de date **array** este constituită din tablouri (tabele). Tablourile sînt formate dintr-un număr fixat de componente de acelaşi tip, denumit **tip de bază**. Referirea componentelor se face cu ajutorul unui **indice**.

Un tip de date *tablou* se defineşte printr-o construcţie de forma

type <*Nume tip*> = **array** [*T1*] **of** *T2*;

unde *T1* este tipul indicelui care trebuie să fie ordinal, iar *T2* este tipul componentelor (tipul de bază) care poate fi un tip oarecare.

Exemple:

```
1)type  Vector = array[1..5] of real;
  var   x: Vector;
2)type  Zi= (L, Ma, Mi, J, V, S, D);
        Venit= array[Zi] of real;
  var   v: Venit;
        z: Zi;
3)type  Ora= 0..23;
        Grade= -40..40;
        Temperatura= array[Ora] of Grade;
  var   t: Temperatura;
        h: Ora;
```

Structura datelor din exemplele în studiu este prezentată în *fig. 4.1.*

Fiecare componentă a unei variabile de tip tablou poate fi specificată explicit, prin numele variabilei urmat de indicele respectiv încadrat de paranteze pătrate.

Exemple:

```
X[1], x[4];
v[L], v [Ma], v[J];
t[0], t[15], t[23];
v[z], t[h].
```

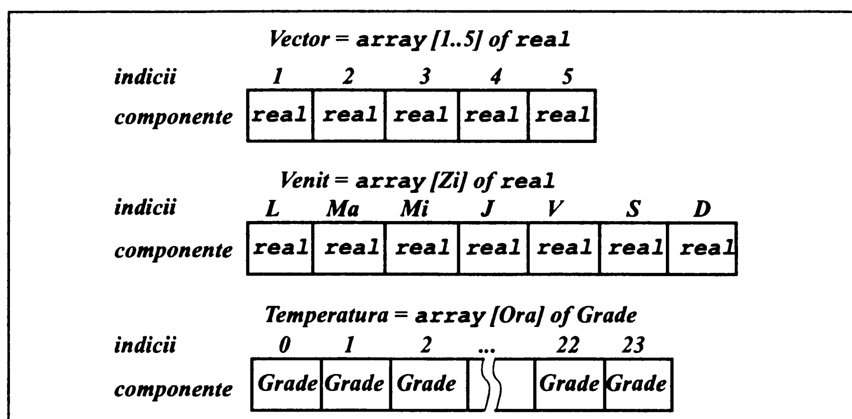


Fig. 4.1. Structura datelor de tip `Vector`, `Venit` şi `Temperatura`

Asupra componentelor datelor de tip *tablou* se pot efectua toate operaţiile admise de tipul de bază respectiv. Programul ce urmează afişează pe ecran suma componentelor variabilei `x` de tip `Vector`. Valorile componentelor `x[1]`, `x[2]`, ..., `x[5]` se citesc de la tastatură.

```
Program P77;
  { Suma componentelor variabilei x de tip Vector }
  type Vector=array[1..5] of real;
  var x : Vector;
      i : integer;
      s : real;
  begin
    writeln('Daţi 5 numere:');
    for i:=1 to 5 do readln(x[i]);
    writeln('Aţi introdus:');
    for i:=1 to 5 do writeln(x[i]);
    s:=0;
    for i:=1 to 5 do s:=s+x[i];
    writeln('Suma=',s);
    readln;
  end.
```

Pentru a extinde aria de aplicare a unui program, se recomandă ca numărul de componente ale datelor de tip **array** să fie specificate prin constante.

De exemplu, programul `P77` poate fi modificat pentru a însuma *n* numere reale, $n \le 100$:

```
Program P78;
{ Extinderea domeniului de aplicare a programului P77 }
  const nmax=100;
```

```
type Vector=array[ 1..nmax] of real;
var x : Vector;
    n : 1..nmax;
    i : integer;
    s : real;
begin
  write('n='); readln(n);
  writeln('Daţi ', n, ' numere:');
  for i:=1 to n do readln(x[ i] );
  writeln('Aţi introdus:');
  for i:=1 to n do writeln(x[ i] );
  s:=0;
  for i:=1 to n do s:=s+x[ i] ;
  writeln('Suma=', s);
  readln;
end.
```

Tablourile bidimensionale se definesc cu ajutorul construcţiei

type <*Nume tip*> = **array**[*T1*, *T2*] **of** *T3*,

unde *T1* şi *T2* specifică tipul indicilor, iar *T3* tipul componentelor.

Pentru exemplificare, în *fig. 4.2.* este prezentată structura datelor tipului:

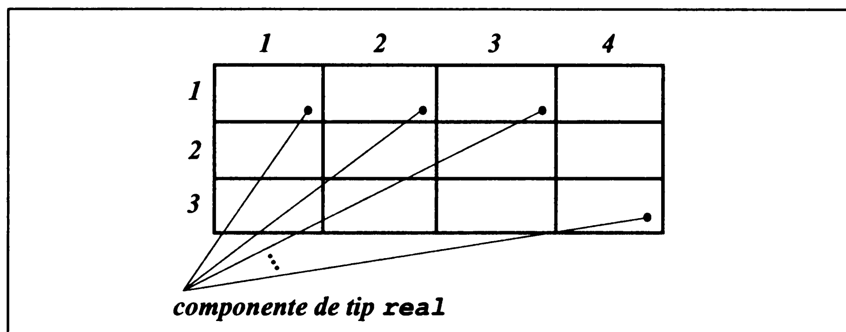


Fig. 4.2. Structura datelor de tip `Matrice`

`Matrice = array[ 1..3, 1..4] of real.`

Componentele unei variabile de tip *tablou* bidimensional se specifică explicit prin numele variabilei urmat de indicii respectivi separaţi prin virgulă şi încadraţi de paranteze pătrate.

De exemplu, în prezenţa declaraţiei

`var m : Matrice;`

notaţia `m[ 1,1]` specifică componenta din linia 1, coloana 1 (vezi *fig. 4.2.*); notaţia `m[ 1,2]` specifică componenta din linia 1, coloana 2; notaţia `m[ i,j]` specifică componenta din linia `i`, coloana `j`.

Programul ce urmează afişează pe ecran suma componentelor variabilei `m` de tip `Matrice`. Valorile componentelor `m[1,1]`, `m[1,2]`, ..., `m[3,4]` se citesc de la tastatură.

```
Program P79;
  { Suma componentelor variabilei m de tip Matrice }
  type Matrice=array[1..3, 1..4] of real;
  var m : Matrice;
      i, j : integer;
      s : real;
  begin
    writeln('Daţi componentele m[i,j]:');
    for i:=1 to 3 do
      for j:=1 to 4 do
      begin
        write('m[', i, ',', j, ']=');
        readln(m[i,j]);
      end;
    writeln('Aţi introdus:');
    for i:=1 to 3 do
      begin
      for j:=1 to 4 do
        write(m[i,j]);
        writeln;
      end;
  s:=0;
  for i:=1 to 3 do
    for j:=1 to 4 do
      s:=s+m[i,j];
    writeln('Suma=', s);
    readln;
  end.
```

În general, un tip **tablou *n*-dimensional** (*n* = 1, 2, 3 ş.a.m.d) se defineşte cu ajutorul diagramelor sintactice din *fig. 4.3*. Atributul **packed** (împachetat) indică cerinţa de optimizare a spaţiului de memorie pentru elementele tipului **array**. Menţionăm că în majoritatea compilatoarelor actuale utilizarea acestui atribut nu are nici un efect, întrucît optimizarea se efectuează în mod automat.

Fiind date două variabile de tip *tablou* de acelaşi tip, numele variabilelor pot apărea într-o instrucţiune de atribuire. Această atribuire înseamnă copierea tuturor componentelor din membrul drept în membrul stîng.

De exemplu, în prezenţa declaraţiilor

```
var a, b : Matrice;
```

instrucţiunea

```
a:=b
```

este corectă.

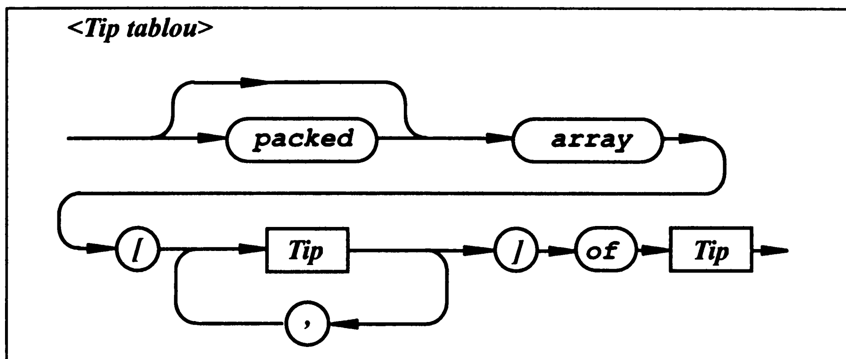


Fig. 4.3. Diagrama sintactică *<Tip tablou>*

În exemplele de mai sus tipul de bază (tipul componentelor) a fost de fiecare dată un tip simplu. Deoarece tipul de bază poate fi, în general, un tip arbitrar, devine posibilă definirea tablourilor cu componente de tip structurat. Considerăm acum un exemplu în care tipul de bază este el însuşi un tip **array**:

```
Type  Linie=array[1..4] of real;
      Tabel=array[1..3] of Linie;
      var L : Linie;
          T : Tabel;
          x : real;
```

Variabila `T` este formată din 3 componente: `T[1]`, `T[2]` şi `T[3]` de tipul `Linie`.

Variabila `L` este formată din 4 componente: `L[1]`, `L[2]`, `L[3]` şi `L[4]` de tipul `real`.

Prin urmare, atribuirile

```
L[1]:=x;  x:=L[3];  T[2]:=L;  L:=T[1]
```

sînt corecte.

Elementele variabilei `T` pot fi specificate prin `T[i][j]` sau prescurtat `T[i,j]`. Aici `i` indică numărul componentei de tip `Linie` în cadrul variabilei `T`, iar `j` — numărul componentei de tip `real` în cadrul componentei `T[i]` de tip `Linie`.

Subliniem faptul că declaraţiile de forma

array[T_1 , T_2] **of** T_3

şi

array[T_1] **of** **array**[T_2] **of** T_3

definesc tipuri distincte de date. Prima declaraţie defineşte tablouri bidimensionale cu componente de tipul T_3 . A doua declaraţie defineşte tablouri unidimensionale cu componente de tipul **array**[T_2] **of** T_3 .

În programele PASCAL tablourile se utilizează pentru a grupa sub un singur nume mai multe variabile cu caracteristici identice.

Întrebări şi exerciţii

❶ Precizaţi tipul indicilor şi tipul componentelor din următoarele declaraţii:

```
type  P=array[ 1..5] of integer;
      Culoare=(Galben, Verde, Albastru, Violet);
      R=array[ Culoare] of real;
      S=array[ Culoare, 1..3] of boolean;
      T=array[ boolean] of Culoare;
```

Reprezentaţi structura datelor de tipul P, R, S şi T pe un desen (*fig. 4.1* şi *4.2*).

❷ Indicaţi pe diagrama sintactică din *fig. 4.3* drumurile care corespund declaraţiilor din exerciţiul 1.

❸ Scrieţi formulele metalingvistice care corespund diagramei sintactice <*Tip tablou*> din *fig. 4.3*.

❹ Se consideră declaraţiile:

```
type Vector= array[ 1..5] of real;
var  x, y : Vector;
```

Scrieţi expresia aritmetică a cărei valoare este:

a) suma primelor trei componente ale variabilei x;
b) suma tuturor componentelor variabilei y;
c) produsul tuturor componentelor variabilei x;
d) valoarea absolută a componentei a treia a variabilei y;
e) suma primelor componente ale variabilelor x şi y.

❺ Se consideră declaraţiile

```
type  Zi= (L, Ma, Mi, J, V, S, D);
      Venit= array[ Zi] of real;
var   v : Venit;
```

Componentele variabilei v reprezintă venitul zilnic al unei întreprinderi. Elaboraţi un program care:

a) calculează venitul săptămînal al întreprinderii;
b) calculează media venitului zilnic;
c) indică ziua în care s-a obţinut cel mai mare venit;
d) indică ziua cu venitul cel mai mic.

❻ Se consideră declaraţiile:

```
type  Ora= 0..23;
      Grade= -40..40;
      Temperatura= array[ Ora] of Grade;
var   t: Temperatura;
```

Componentele variabilei t reprezintă temperaturile măsurate din oră în oră pe parcursul a 24 de ore. Elaboraţi un program care:

a) calculează temperatura medie;
b) indică maximul şi minimul temperaturii;
c) indică ora (orele) la care s-a înregistrat temperatura maximă;
d) indică ora (orele) la care s-a înregistrat temperatura minimă.

❼ Se consideră declaraţiile

```
type  Oras = (Chisinau, Orhei, Balti,
              Tighina, Tiraspol);
      Zi = (L, Ma, Mi, J, V, S, D);
      Consum = array[Oras, Zi] of real;
var   c : Consum;
      or : Oras;
      z : Zi;
```

Componenta C[or,z] a variabilei C reprezintă consumul de energie electrică a oraşului or în ziua z. Elaboraţi un program care:

a) calculează energia electrică consumată de fiecare oraş pe parcursul unei săptămîni;
b) calculează energia electrică consumată zilnic de oraşele în studiu;
c) indică oraşul cu un consum săptămînal maxim;
d) indică oraşul cu un consum săptămînal minim;
e) indică ziua în care se consumă cea mai multă energie electrică;
f) indică ziua cu un consum minim.

❽ Se consideră declaraţiile:

```
type  Vector = array[1..5] of real;
      Matrice= array[1..3,1..4] of real;
      Linie= array[1..4] of real;
      Tabel= array[1..3] of Linie;
var   V : Vector;
      M : Matrice;
      L : Linie;
      T : Tabel;
      x : real;
      i : integer;
```

Care din atribuirile ce urmează sînt corecte?

a) `T[3]:=T[1];`
b) `M:=T;`
c) `L:=V;`
d) `L[3]:=x;`
e) `x:=i;`
f) `i:=x;`
g) `L[3]:=i;`
h) `i:=M[1,2];`
i) `x:=V[4];`
l) `T[2]:=V;`
m) `L:=T[3];`
n) `T[1,2]:=M[1,2];`
o) `T[1,2]:=M[1,2];`
p) `M[1]:=4;`
q) `M[1,3]:=L[2];`
r) `x:=T[1][2];`
s) `x:=M[1];`
t) `L:=M[1];`

```
    NP:=N+' '+P;   L:=length(NP); writeln(NP, L:4);
    readln;
  end.
```

Rezultatele afişate pe ecran:

```
Munteanu   8
Mihai   5
Munteanu Mihai  14
Olaru   5
Ion   3
Olaru Ion 9
```

Se observă că pe parcursul derulării programului în studiu lungimea şirurilor de caractere N, P şi NP se schimbă.

Asupra şirurilor de caractere sînt admise operaţiile relaţionale <, <=, =, >=, >, <>. Şirurile se compară componentă cu componentă de la stînga la dreapta în conformitate cu ordonarea caracterelor în tipul de date `char` (vezi paragraful 2.5). Ambii operanzi trebuie să fie de tip `packed array` [1..n] `of char` cu acelaşi număr de componente sau de tip **`string`**. Evident, operanzii de tip **`string`** pot avea lungimi arbitrare.

De exemplu, rezultatul operaţiei *'AC'<'BA'* este `true`, iar rezultatul operaţiei *'AAAAC'<'AAAAB'* este `false`.

O variabilă de tip *şir de caractere* poate fi folosită fie în totalitatea ei, fie parţial, prin referirea unui caracter din şir.

De exemplu, pentru P=*'Mihai'* avem P[1]=*'M'*, P[2]='i', P[3]=*'h'* ş.a.m.d. După executarea secvenţei de instrucţiuni

```
P[1]:='P'; P[2]:='e'; P[3]:='t';
P[4]:='r'; P[5]:='u'
```

variabila P va avea valoarea *'Petru'*.

Programul ce urmează citeşte de la tastatură şiruri arbitrare de caractere şi afişează pe ecran numărul de spaţii în şirul respectiv. Derularea programului se termină după introducerea şirului *'Sfîrşit'*.

```
Program P82;
  { Numărul de spaţii într-un şir de caractere }
  var S : string;
      i,j : integer;
  begin
    writeln('Daţi şiruri de caractere:');
    repeat
    readln(S);
    i:=0;
    for j:=1 to length(S) do
    if S[j]=' ' then i:=i+1;
    writeln('Numărul de spaţii=', i);
```

```
  until S='Sfîrşit';
end.
```

Întrebări şi exerciţii

❶ Cum se defineşte un tip de date *şir de caractere*?
❷ Ce operaţii pot fi efectuate asupra şirurilor de caractere?
❸ Comentaţi următorul program:

```
Program P83;
  { Eroare}
  var S : packed array[ 1..5] of char;
  begin
    S:='12345';
    writeln(S);
    S:='Sfat';
    writeln(S);
  end.
```

❹ Elaboraţi un program care:

a) determină numărul de apariţii ale caracterului 'A' într-un şir;
b) substituie caracterul 'A' prin caracterul '*';
c) radiază din şir caracterul 'B';
d) determină numărul de apariţii al silabei 'MA' într-un şir;
e) substituie silabele 'MA' prin silaba 'TA';
f) radiază din şir silaba 'TO';

❺ Precizaţi rezultatul operaţiilor relaţionale:

a) 'B' < 'A';
b) 'BB' > 'AA';
c) 'BAAAA' < 'AAAAA';
d) 'CCCCD' > 'CCCCA';
e) 'A A' = 'AA';
f) 'BB ' < 'B B';
g) 'A' = 'a';
h) 'Aa' > 'aA';
i) '123' = '321';
j) '12345' > '12345'.

❻ Se consideră şiruri de caractere formate din literele mari ale alfabetului latin şi spaţii. Elaboraţi un program care afişează şirurile în studiu după următoarele reguli:
– fiecare literă de la 'A' pînă la 'Y' se înlocuieşte prin următoarea literă din alfabet;
– fiecare literă 'Z' se înlocuieşte prin litera 'A';
– fiecare spaţiu se înlocuieşte prin '-'.

❼ Elaboraţi un program care descifrează şirurile cifrate conform regulilor din exerciţiul 6.

❽ Se consideră m, $m \leq 100$, şiruri de caractere formate din literele mici ale alfabetului latin. Elaboraţi un program care afişează pe ecran şirurile în studiu în ordine alfabetică.

❾ Şirul S este compus din cîteva propoziţii, fiecare terminîndu-se cu punct, semn de exclamare sau semnul întrebării. Elaboraţi un program care afişează pe ecran numărul de propoziţii din şirul în studiu.

```
    NP:=N+' '+P;   L:=length(NP); writeln(NP, L:4);
    readln;
  end.
```

Rezultatele afişate pe ecran:

```
Munteanu  8
Mihai  5
Munteanu Mihai  14
Olaru  5
Ion  3
Olaru Ion 9
```

Se observă că pe parcursul derulării programului în studiu lungimea şirurilor de caractere N, P şi NP se schimbă.

Asupra şirurilor de caractere sînt admise operaţiile relaţionale <, <=, =, >=, >, <>. Şirurile se compară componentă cu componentă de la stînga la dreapta în conformitate cu ordonarea caracterelor în tipul de date `char` (vezi paragraful 2.5). Ambii operanzi trebuie să fie de tip `packed array` [1..n] `of char` cu acelaşi număr de componente sau de tip **`string`**. Evident, operanzii de tip **`string`** pot avea lungimi arbitrare.

De exemplu, rezultatul operaţiei *'AC'<'BA'* este `true`, iar rezultatul operaţiei *'AAAAC'<'AAAAB'* este `false`.

O variabilă de tip *şir de caractere* poate fi folosită fie în totalitatea ei, fie parţial, prin referirea unui caracter din şir.

De exemplu, pentru `P='Mihai'` avem `P[1]='M'`, `P[2]='i'`, `P[3]='h'` ş.a.m.d. După executarea secvenţei de instrucţiuni

```
P[1]:='P'; P[2]:='e'; P[3]:='t';
P[4]:='r'; P[5]:='u'
```

variabila P va avea valoarea *'Petru'*.

Programul ce urmează citeşte de la tastatură şiruri arbitrare de caractere şi afişează pe ecran numărul de spaţii în şirul respectiv. Derularea programului se termină după introducerea şirului *'Sfîrşit'*.

```
Program P82;
  {Numărul de spaţii într-un şir de caractere}
  var S : string;
      i,j : integer;
  begin
    writeln('Daţi şiruri de caractere:');
    repeat
    readln(S);
    i:=0;
    for j:=1 to length(S) do
    if S[j]=' ' then i:=i+1;
    writeln('Numărul de spaţii=', i);
```

```
  until S='Sfîrşit';
end.
```

Întrebări şi exerciţii

❶ Cum se defineşte un tip de date *şir de caractere*?
❷ Ce operaţii pot fi efectuate asupra şirurilor de caractere?
❸ Comentaţi următorul program:

```
Program P83;
 { Eroare}
 var S : packed array[ 1..5] of char;
 begin
   S:='12345';
   writeln(S);
   S:='Sfat';
   writeln(S);
 end.
```

❹ Elaboraţi un program care:

a) determină numărul de apariţii ale caracterului 'A' într-un şir;
b) substituie caracterul 'A' prin caracterul '*';
c) radiază din şir caracterul 'B';
d) determină numărul de apariţii al silabei 'MA' într-un şir;
e) substituie silabele 'MA' prin silaba 'TA';
f) radiază din şir silaba 'TO';

❺ Precizaţi rezultatul operaţiilor relaţionale:

a) 'B' < 'A';
b) 'BB' > 'AA';
c) 'BAAAA' < 'AAAAA';
d) 'CCCCD' > 'CCCCA';
e) 'A A' = 'AA';
f) 'BB ' < 'B B';
g) 'A' = 'a';
h) 'Aa' > 'aA';
i) '123' = '321';
j) '12345' > '12345'.

❻ Se consideră şiruri de caractere formate din literele mari ale alfabetului latin şi spaţii. Elaboraţi un program care afişează şirurile în studiu după următoarele reguli:
– fiecare literă de la 'A' pînă la 'Y' se înlocuieşte prin următoarea literă din alfabet;
– fiecare literă 'Z' se înlocuieşte prin litera 'A';
– fiecare spaţiu se înlocuieşte prin '-'.

❼ Elaboraţi un program care descifrează şirurile cifrate conform regulilor din exerciţiul 6.

❽ Se consideră *m*, $m \leq 100$, şiruri de caractere formate din literele mici ale alfabetului latin. Elaboraţi un program care afişează pe ecran şirurile în studiu în ordine alfabetică.

❾ Şirul S este compus din cîteva propoziţii, fiecare terminîndu-se cu punct, semn de exclamare sau semnul întrebării. Elaboraţi un program care afişează pe ecran numărul de propoziţii din şirul în studiu.

4.3. TIPURI DE DATE *ARTICOL* (RECORD)

Mulţimea de valori ale unui tip de date **record** este constituită din articole (înregistrări). Articolele sînt formate din componente, denumite ***cîmpuri***. Spre deosebire de componentele unui tablou, cîmpurile pot fi de tipuri diferite. Fiecare cîmp are un nume (identificator de cîmp).

Un tip de date *articol* se defineşte printr-o structură de forma

```
type <Nume tip> = record
                   <Nume cîmp 1> : T1;
                   <Nume cîmp 2> : T2;
                   ...
                   <Nume cîmp n> : Tn;
                   end;
```

unde T_1, T_2, .., T_n specifică tipul cîmpurilor respective. Tipul unui nume de cîmp este arbitrar, astfel un cîmp poate să fie la rîndul lui tot de tip *articol*. Prin urmare, se pot defini tipuri imbricate.

Exemple:

```
1)  type  Elev=record
                Nume : string;
                Prenume : string;
                NotaMedie : real;
                end;
  Var   E1, E2 : Elev;
2)  type    Punct=record
                  x : real; { coordonata x}
                  y : real; { coordonata y}
                  end;
  var   P1, P2 : Punct;
3)  type    Triunghi = record
                        A : Punct; { vîrful A}
                        B : Punct; { vîrful B}
                        C : Punct; { vîrful C}
                        end;
  var   T1, T2, T3 : Triunghi;
```

Structura datelor din exemplele în studiu este prezentată în *fig. 4.4*.

Fiind date două variabile de tip *articol* de acelaşi tip, numele variabilelor pot apărea într-o instrucţiune de atribuire. Această atribuire înseamnă copierea tuturor cîmpurilor din membrul drept în membrul stîng. De exemplu, pentru tipurile de date şi variabile declarate mai sus instrucţiunile

```
E1:=E2;
T2:=T3;
P2:=P1
```

sînt corecte.

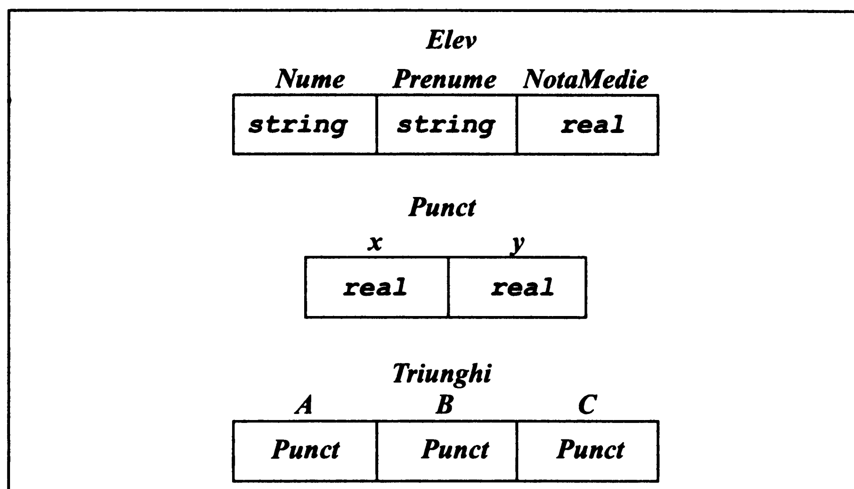


Fig. 4.4. Structura datelor de tip `Elev`, `Punct` şi `Triunghi`

Fiecare componentă a unei variabile de tip **record** poate fi specificată explicit, prin numele variabilei şi denumirile de cîmpuri, separate prin puncte.

Exemple:

```
E1.Nume, E1.Prenume, E1.NotaMedie;
E2.Nume, E2.Prenume, E2.NotaMedie;
P1.x, P1.y, P2.x, P2.y;
T1.A, T1.B, T1.C, T2.A, T2.B, T2.C,
T1.A.x, T1.A.y, T2.B.x, T2.B.y;
```

Evident, componenta `E1.Nume` este de tip **string**; componenta `P1.x` este de tip `real`; componenta `T1.A` este de tip `Punct`; componenta `T1.A.x` este de tip `real` ş.a.m.d.

Asupra componentelor datelor de tip *articol* se pot efectua toate operaţiile admise de tipul cîmpului respectiv. Programul ce urmează compară notele medii a doi elevi şi afişează pe ecran numele şi prenumele elevului cu nota medie mai bună. Se consideră că elevii au note medii diferite.

```
Program P84;
  { Date de tipul Elev }
 type Elev=record
               Nume : string;
               Prenume : string;
               NotaMedie : real
            end;
 var E1, E2, E3 : Elev;
 begin
```

```
  writeln('Daţi datele primului elev:');
  write('Numele:  '); readln(E1.Nume);
  write('Prenumele:  '); readln(E1.Prenume);
  write('Nota medie:  '); readln(E1.NotaMedie);

  writeln('Daţi datele elevului al doilea:  ');
  write('Numele:  '); readln(E2.Nume);
  write('Prenumele:  '); readln(E2.Prenume);
  write('Nota medie:  '); readln(E2.NotaMedie);

   if E1.NotaMedie > E2.NotaMedie
   then E3:=E1 else E3:=E2;

  writeln('Elevul cu media mai bună:');
  writeln(E3.Nume, ' ', E3.Prenume,
         E3.NotaMedie : 5:2);
   readln;
 end.
```

Orice tip de date **record** poate servi ca tip de bază pentru formarea altor tipuri structurate.

Exemplu:

```
type ListaElevilor= array[1..40] of Elev;
var  LE: ListaElevilor;
```

Evident, notaţia `LE[i]` specifică elevul `i` din listă; notaţia `LE[i].Nume` specifică numele acestui elev ş.a.m.d. Programul ce urmează citeşte de la tastatură datele referitoare la `n` elevi şi afişează pe ecran numele, prenumele şi nota medie a celui mai bun elev. Se consideră că elevii au note medii diferite.

```
Program P85;
  { Tablou cu componente de tipul Elev }
 type Elev = record
               Nume : string;
               Prenume : string;
               NotaMedie : real
            end;
     ListaElev = array[1..40] of Elev;
  Var E : Elev;
     LE : ListaElev;
     n : 1..40;
     i : integer;
 begin
  write('n='); readln(n);
  for i:=1 to n do
    begin
     writeln('Daţi datele elevului ', i);
```

```
      write('Numele:  '); readln(LE[ i] .Nume);
      write('Prenumele:  ');
      readln(LE[ i] .Prenume);
      write('Nota Medie:  ');
      readln(LE[ i] .NotaMedie);
    end;
  E.NotaMedie:=0;
  for i:=1 to n do
    if LE[ i] .NotaMedie > E.NotaMedie
      then E:=LE[ i] ;
  writeln('Cel mai bun elev:');
  writeln(E.Nume,' ',E.Prenume,E.NotaMedie : 5:2);
  readln;
end.
```

În general, un tip de date *articol* se defineşte cu ajutorul diagramelor sintactice din *fig. 4.5*. În completare la articolele cu un număr fix de cîmpuri, limbajul PASCAL permite utilizarea articolelor cu variante. Aceste tipuri de date se studiază în cursurile avansate de informatică.

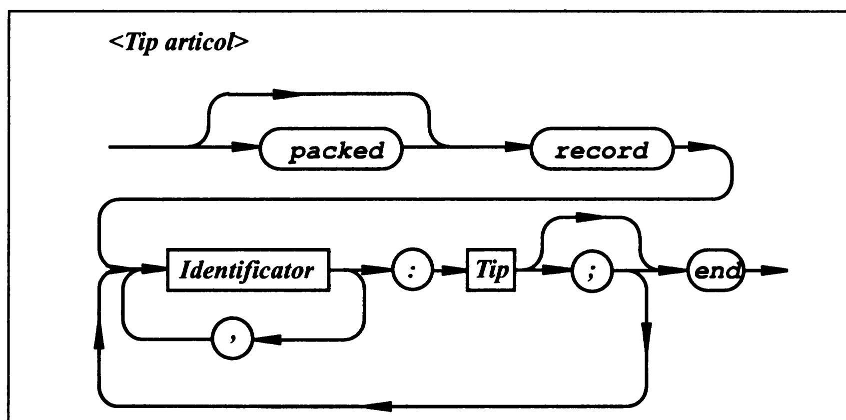


Fig. 4.5. Diagrama sintactică <*Tip articol*>

Întrebări şi exerciţii

❶ Care este mulţimea de valori ale unui tip de date *articol*?

❷ Indicaţi pe diagrama sintactică din *fig. 4.5* drumurile care corespund definiţiilor tipurilor de date *articol* din programele P84 şi P85.

❸ Scrieţi formulele metalingvistice pentru diagrama sintactică din *fig. 4.5*.

❹ Se consideră următoarele tipuri de date

```
type  Data= record
              Ziua : 1..31;
              Luna : 1..12;
```

```
        Anul : integer;
        end;
    Persoana=record
        NumePrenume : string;
        DataNasterii : Data
        end;
    ListaPersoane=array [1..50] of Persoana;
```

Elaboraţi un program care citeşte de pe tastatură datele referitoare la n persoane ($n \le 50$) şi afişează pe ecran:

a) persoanele născute în ziua z a lunii;
b) persoanele născute în luna l a anului;
c) persoanele născute în anul a;
d) persoanele născute pe data $z.l.a$;
e) persoana cea mai în vîrstă;
f) persoana cea mai tînără;
g) vîrsta fiecărei persoane în ani, luni, zile;
h) lista persoanelor care au mai mult de v ani;
i) lista persoanelor în ordine alfabetică;
j) lista persoanelor ordonată conform datei naşterii;
k) lista persoanelor de aceeaşi vîrstă (născuţi în acelaşi an).

❺ Se consideră n puncte ($n \le 30$) pe un plan euclidian. Fiecare punct i este definit prin coordonatele sale x_i, y_i. Distanţa dintre punctele i şi j se calculează după formula

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Elaboraţi un program care afişează pe ecran punctele distanţa dintre care este maximă.

❻ Aria triunghiului este dată de formula lui Heron

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

unde p este semiperimetrul, iar a, b şi c sînt lungimile laturilor respective. Utilizînd tipurile de date `Punct` şi `Triunghi` din paragraful în studiu, elaboraţi un program care citeşte de la tastatură informaţiile referitoare la n triunghiuri ($n \le 10$) şi afişează pe ecran:

a) aria fiecărui triunghi;
b) coordonatele vîrfurilor triunghiului cu aria maximă;
c) coordonatele vîrfurilor triunghiului cu aria minimă;
d) informaţiile referitoare la fiecare triunghi în ordinea creşterii ariilor.

4.4. INSTRUCŢIUNEA WITH

Componentele unei variabile de tip articol se specifică explicit prin numele variabilei şi denumirile de cîmpuri, separate prin puncte.

De exemplu, în prezenţa declaraţiilor

```
type  Angajat=record
                NumePrenume : string;
                ZileLucrate : 1..31;
                PlataPeZi   : real;
                PlataPeLuna : real;
                end;
 var A : angajat;
```

componentele variabilei A se specifică prin A.NumePrenume, A.ZileLucrate, A.PlataPeZi şi A.PlataPeLuna.

Întrucît numele A al variabilei de tip articol se repetă de mai multe ori, acest mod de referire a componentelor este în anumite situaţii greoi. Repetările obositoare pot fi evitate cu ajutorul instrucţiunii **with** (cu).

Sintaxa instrucţiunii în studiu este:

<*Instrucţiune with*> ::= **with** <*Variabilă*> {, <*Variabilă*>}
do <*Instrucţiune*>

Diagrama sintactică este prezentată în *fig. 4.6*.

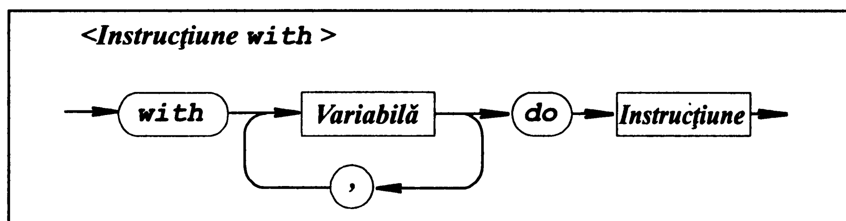


Fig. 4.6. Diagrama sintactică a instrucţiunii with

În interiorul unei instrucţiuni **with** componentele uneia sau a mai multor variabile de tip *articol* pot fi referite folosind numai numele cîmpurilor respective.

Exemplu:

```
with A do PlataPeLuna:=PlataPeZi*ZileLucrate
```

Această instrucţiune este echivalentă cu următoarea:

```
A.PlataPeLuna:=A.PlataPeZi*A.ZileLucrate
```

Utilizarea instrucţiunii **with** necesită o atenţie sporită din partea programatorului, care este obligat să specifice univoc componentele variabilelor de tip *articol*. În interiorul unei instrucţiuni **with**, la întîlnirea unui identificator, prima dată se testează dacă el poate fi interpretat ca un nume de cîmp al articolului respectiv. Dacă da, identificatorul va fi interpretat ca atare, chiar dacă în acel moment este accesibilă şi o variabilă avînd acelaşi nume.

Exemplu:

```
type  Punct=record
              x : real;
              y : real;
              end;
      Segment=record
              A : Punct;
              B : Punct;
              end;
var   P : Punct;
      S : Segment;
      x : integer;
```

În cazul nostru identificatorul x poate să reprezinte fie variabila de tip integer x, fie cîmpul P.x al articolului P.

În instrucțiunea

```
x:=1
```

identificatorul x se referă la variabila de tip integer x.

În instrucțiunea

```
with P do x:=1
```

identificatorul x se referă la cîmpul P.x al variabilei de tip articol P.

Întrucît variabila de tip articol S nu conține nici un cîmp cu numele S.x, în instrucțiunea

```
with S do x:=1
```

identificatorul x va fi interpretat ca variabila de tip integer x.

O instrucțiune de forma

with v_1, v_2, ..., v_n **do** <*Instrucțiune*>,

unde v_1, v_2, ..., v_n sînt variabile de tip *articol*, este echivalentă cu instrucțiunea

with v_1 **do**
with v_2 **do**
{...}
with v_n **do** <*Instrucțiune*>.

Evident, componentele variabilelor v_1, v_2, ..., v_n trebuie specificate univoc prin denumirile cîmpurilor respective.

De exemplu, pentru variabilele P și S, declarate mai sus, putem scrie:

```
with P, S do
begin
  x:=1.0;    {referire la P.x}
  y:=1.0;
  A.x:=0;    {referire la S.A.x}
```

```
  A.y:=0;
  B.x:=2.0; {referire la S.B.x}
  B.y:=2.0;
end;
```

În mod obişnuit, instrucţiunea **with** se utilizează numai în cazurile în care se ajunge la o reducere semnificativă a textului unui program.

Întrebări şi exerciţii

❶ Indicaţi pe diagrama sintactică din *fig. 4.6* drumurile care corespund instrucţiunilor **with** din exemplele paragrafului în studiu.

❷ Care este destinaţia instrucţiunii **with**?

❸ Utilizînd instrucţiunea **with**, excludeţi din programele P84 şi P85 din paragraful precedent repetările de genul

```
El.Nume, El.Prenume, ...,
LE[i].Nume, LE[i].Prenume.
```

❹ Se consideră următoarele tipuri de date:

```
Type  Angajat=record
                NumePrenume : string;
                ZileLucrate : 1..31;
                PlataPeZi : real;
                PlataPeLuna : real;
end;
ListaDePlata=array[1..50] of Angajat;
```

Plata pe lună a fiecărui angajat se calculează înmulţind plata pe zi cu numărul de zile lucrate. Elaboraţi un program care:

a) calculează plata pe lună a fiecărui angajat;
b) calculează salariul mediu al angajaţilor incluşi în listă;
c) afişează pe ecran datele despre angajaţii cu plata lunară maximă;
d) afişează lista angajaţilor ordonată alfabetic;
e) afişează lista angajaţilor în ordinea creşterii plăţilor pe zi;
f) ordonează lista angajaţilor în ordinea creşterii plăţilor pe lună;
g) afişează lista angajaţilor în ordinea creşterii numărului de zile lucrate.

❺ Un cerc poate fi definit prin coordonatele *x*, *y* şi raza *r*. Elaboraţi un program care citeşte de la tastatură datele referitoare la *n* cercuri ($n \leq 50$) şi afişează pe ecran:

a) coordonatele centrului şi raza cercului cu aria maximă;
b) numărul de cercuri incluse în cercul cu raza maximă şi coordonatele centrelor respective;
c) coordonatele centrului şi raza cercului cu aria minimă;
d) numărul de cercuri în care este inclus cercul cu raza minimă şi coordonatele centrelor respective.

4.5. TIPURI DE DATE *MULŢIME* (SET)

Un tip de date *mulţime* (**set**) se defineşte în raport cu un tip de bază care trebuie să fie ordinal:

<*Tip mulţime*> ::= [**packed**] **set of** <*Tip*>

Valorile unui tip de date **set** sînt mulţimi formate din valorile tipului de bază. Dacă tipul de bază are *n* valori, tipul mulţime va avea 2^n valori. În implementările limbajului valoarea lui *n* este limitată, de regulă $n \leq 256$.

În PASCAL o mulţime poate fi specificată enumerîndu-i-se elementele între parantezele pătrate "[" şi "] ", care ţin locul acoladelor din matematică.

Notaţia [ ] reprezintă mulţimea vidă.

Exemple:

```
type  Indice=1..10;
      Zi=(L, Ma, Mi, J, V, S, D);
      MultimeIndicii=set of Indice;
      ZileDePrezenta=set of Zi;
var   MI : MultimeIndicii;
      ZP : ZileDePrezenta;
```

Tipul ordinal `Indice` are n = 10 valori: `1, 2, 3, 4, 5, 6, 7, 8, 9, 10`. Tipul `MultimeIndicii` are 2^{10}= 1024 de valori, şi anume:

```
[ ],[ 1],[ 2],...,[ 1, 2],[ 1, 3],...,
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
```

Prin urmare, variabila `MI` poate să aibă oricare din aceste valori, de exemplu:

```
MI:= [ 1, 3] .
```

Tipul ordinal `Zi` are n = 7 valori: `L, Ma, Mi, J, V, S, D`. Tipul `ZileDePrezenta` are 2^7 = 128 de valori, şi anume

```
[ ],[ L],[ Ma],[ Mi],...,[ L, Ma],[ L, Mi],...,
[ L, Ma, Mi, J, V, S, D].
```

Variabila `ZP` poate să aibă oricare dintre aceste valori, de exemplu

```
ZP := [ L, Ma, Mi, V] .
```

O valoare de tip *mulţime* poate fi specificată printr-un **constructor** (generator) de mulţime. Diagrama sintactică a unităţii gramaticale <*Constructor mulţime*> este prezentată în *fig. 4.7*. Un constructor conţine specificarea elementelor mulţimii, separate prin virgule, şi incluse între paranteze pătrate. Un element poate să fie o valoare concretă a tipului de bază sau un interval de forma:

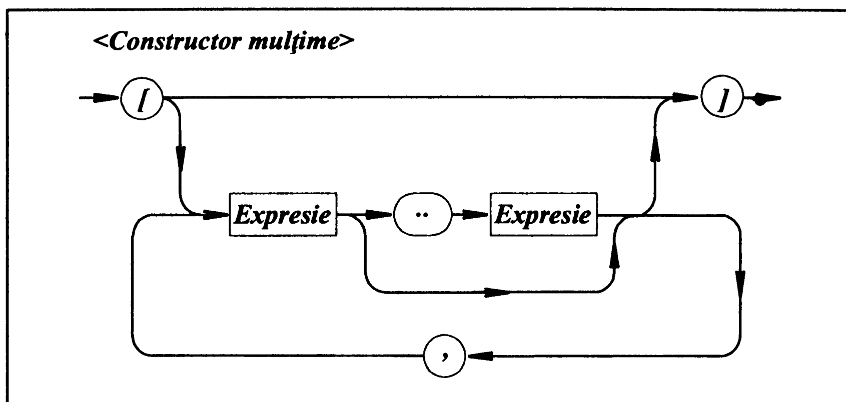


Fig. 4.7. Diagrama sintactică *<Constructor mulţime>*

<Expresie> . . *<Expresie>*

Valorile expresiilor în studiu precizează limitele inferioare şi superioare ale intervalului.

Exemple :

```
[]
[ 1, 2, 3, 8]
[ 1..4, 8..10]
[ i-k..i+k]
[ L, Ma, V..D]
```

Asupra valorilor unui tip de date *mulţime* se pot efectua operaţiile uzuale:

+ reuniunea;
* intersecţia;
\- diferenţa,

rezultatul fiind de tip *mulţime* şi operaţiile relaţionale:

= egalitatea;
<> inegalitatea;
<=, => incluziunea;
in apartenenţa,

rezultatul fiind de tip `boolean`.

Programul ce urmează afişează pe ecran rezultatele operaţiilor +, * şi -, efectuate asupra valorilor de tip `MultimeIndicii`.

```
Program P86;
  { Date de tip MultimeIndicii }
 type Indice=1..10;
```

```
    MultimeIndicii=set of Indice;
var  A, B, C : MultimeIndicii;
     i : integer;
begin
  A:=[ 1..5, 8] ;      { A conţine 1,2,3,4,5,8 }
  B:=[ 1..3, 9, 10] ; { B conţine 1,2,3,9,10 }
  C:=[ ] ;             { C este o mulţime vidă }

  C:=A+B;              { C conţine 1,2,3,4,5,8,9,10 }
  writeln('Reuniune');
  for i:=1 to 10 do
    if i in C then write(i:3);
  writeln;

  C:=A*B;              { C conţine 1, 2, 3 }
  writeln('Intersecţie');
  for i:=1 to 10 do
    if i in C then write(i:3);
  writeln;
  C:=A-B;              { C conţine 4, 5, 8 }
  writeln('Diferenţa');
  for i:=1 to 10 do
    if i in C then write(i:3);
  writeln;
  readln;
end.
```

Spre deosebire de tablouri şi articole, componentele cărora pot fi referite direct, respectiv prin indicii şi denumiri de cîmpuri, elementele unei mulţimi nu pot fi referite. Se admite numai verificarea apartenenţei elementului la o mulţime (operaţia relaţională **in**). În pofida acestui fapt, utilizarea tipurilor de date *mulţime* măreşte viteza de execuţie şi îmbunătăţeşte lizibilitatea programelor PASCAL. De exemplu, instrucţiunea:

```
if (c='A') or (c='E') or (c='I')
           or (c='O') or (c='U') then ...
```

poate fi înlocuită cu o instrucţiune mai simplă:

```
if c in [ 'A','E','I','O','U'] then ...
```

Un alt exemplu sugestiv este utilizarea tipurilor de date *mulţime* în calcularea numerelor prime mai mici decît un număr natural dat *n*. Pentru aceasta se foloseşte algoritmul *Ciurul (sita) lui Eratostene:*

1) în sită se depun numerele 2, 3, 4, ..., *n*;
2) din sită se extrage cel mai mic număr *i*;
3) numărul extras se include în mulţimea numerelor prime;
4) din sită se elimină toţi multiplii *m* ai numărului *i*;
5) procesul se încheie cînd sita s-a golit .

```
Program P87;
  { Ciurul (sita) lui Eratostene }
 const n=50;
 type MultimeDeNumere=set of 1..n;
 var Sita, NumerePrime : MultimeDeNumere;
                   i, m : integer;
 begin
 { 1} Sita:=[ 2..n] ;
      NumerePrime:=[ ] ;
      i:=2;
      repeat
 { 2}    while not (i in Sita) do i:=succ(i);
 { 3}    NumerePrime:=NumerePrime+[ i] ;
         write(i:4);
         m:=i;
 { 4}    while m<=n do
              begin Sita:=Sita-[ m] ; m:=m+i; end;
 { 5} until Sita=[ ] ;
      writeln;
      readln;
 end.
```

Corespondenţa dintre punctele algoritmului şi instrucţiunile care le exprimă este indicată în comentariile din partea stîngă a liniilor de program.

Întrebări şi exerciţii

❶ Enumeraţi valorile posibile ale variabilelor din declaraţiile ce urmează:

```
var V : set of 'A'..'C';
    S : set of (A, B, C);
    I : set of '1'..'2';
    J : set of 1..2;
```

❷ Comentaţi următorul program :

```
Program P88;
 { Eroare }
 type Multime=set of integer;
 var  M : Multime;
      i : integer;
 begin
  M:=[ 1, 8, 13] ;
  for i:=1 to MaxInt do
    if i in M then writeln(i);
 end.
```

❸ Se consideră următoarele declaraţii:

```
type Culoare=(Galben,Verde,Albastru,Violet);
     Nuanta = set of Culoare;
var  NT : Nuanta;
```

Care sînt valorile posibile ale variabilei NT?

❹ Scrieţi formula metalingvistică care corespunde diagramei sintactice <*Constructor mulţime*> din *fig. 4.7.*

❺ Se consideră tipul de date MultimeIndicii din paragraful în studiu. Precizaţi mulţimile specificate de constructorii ce urmează:

a) [ ] ;	f) [4..3] ;
b) [1..10] ;	g) [1..3, 7..6, 9] ;
c) [1..3, 9..10] ;	h) [4-2..7+1] ;
d) [1+1, 4..7, 9] ;	i) [7-5..4+4] ;
e) [3, 7..9] ;	j) [6, 9, 1..2] .

❻ Elaboraţi un program care afişează pe ecran toate submulţimile mulţimii {1, 2, 3, 4}.

❼ Elaboraţi un program care afişează pe ecran toate submulţimile mulţimii {*'A'*, *'B'*, *'C'*, *'D'*}.

❽ Se dă un şir de caractere în care cuvintele sînt separate fie prin spaţiu, fie prin caracterele *punct, virgulă, punct şi virgulă, semnul exclamării* şi *semnul întrebării*. Elaboraţi un program care afişează pe ecran cuvintele şirului de caractere citit de la tastatură.

❾ Se dă un şir de caractere. Elaboraţi un program care afişează pe ecran numărul de vocale din şir.

❿ Elaboraţi un program care citeşte de la tastatură două şiruri de caractere şi afişează pe ecran :

a) caracterele care se întîlnesc cel puţin în unul din şiruri;
b) caracterele care apar în ambele şiruri;
c) caracterele care apar în primul şi nu apar în şirul al doilea.

⓫ Să se scrie un program care verifică dacă numele unei persoane este introdus corect (numele este un şir de caractere ce nu conţine cifre).

⓬ În implementările actuale ale limbajului numărul valorilor tipului de bază al unui tip *mulţime* este limitat , obişnuit $n \leq 256$. În consecinţă, programul P87 nu poate calcula numere prime mai mari decît n. Elaboraţi un program pentru calcularea numerelor prime din intervalul 8, ..., 10 000.
Indicaţie: Sita din algoritmul lui Eratostene poate fi reprezentată printr-un tablou, componentele căruia sînt mulţimi.

4.6. GENERALITĂŢI DESPRE FIŞIERE

Prin **fişier** se înţelege o structură de date care constă dintr-o secvenţă de componente. Fiecare componentă din secvenţă are acelaşi tip, denumit *tip de bază*. Numărul componentelor din secvenţă nu este fixat, însă sfîrşitul secvenţei este indicat de un simbol special, notat *EOF* (*End of File* — sfîrşit de fişier). Fişierul care nu conţine nici o componentă se numeşte *fişier vid*.

Un tip de date fişier se defineşte printr-o declaraţie de forma:

<Tip fişier> ::= [**packed**] **file of** *<Tip>* ;

unde *<Tip>* este tipul de bază. Tipul de bază este un tip arbitrar, exceptînd tipul *fişier* (nu există "fişier de fişiere").

Exemple:

```
1)type  FisierNumere=file of integer;
  var   FN : FisierNumere;
        n : integer;
2)type  FisierCaractere=file of char;
  var   FC : FisierCaractere;
        c : char;
3)type  Elev=record
            Nume : string;
            Prenume: string;
            NotaMedie : real;
            end;
        FisierElevi=file of Elev;
  var   FE : FisierElevi
        E : Elev;
```

Structura datelor în studiu este prezentată în *fig. 4.8*. Subliniem faptul că elementul *EOF*, care indică sfîrşitul secvenţei nu este o componentă a fişierului.

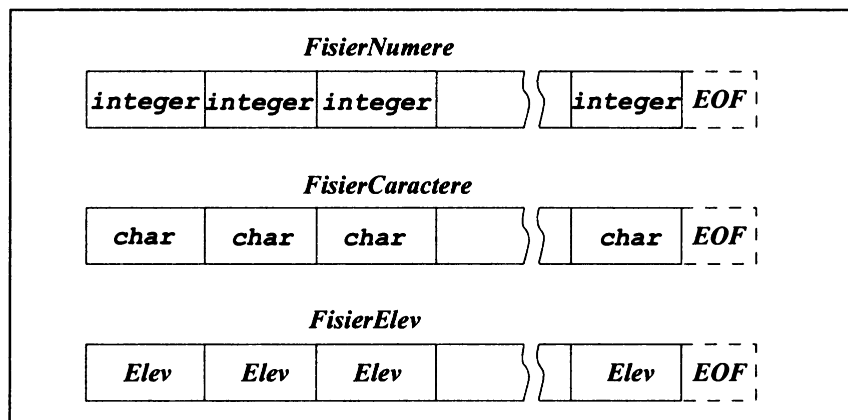


Fig. 4.8. Structura datelor de tip FisierNume, FisierCaractere şi FisierElevi

Variabilele FN, FC, FE ş.a. de tip *fişier* se numesc **fişiere PASCAL** sau, pur şi simplu, *fişiere*. Spre deosebire de celelalte tipuri de date, valorile cărora se păstrează în memoria internă a calculatorului, datele fişierelor PASCAL se păstrează pe suporturile de informaţie ale echipamentelor periferice (discuri şi benzi magnetice, discuri optice, hîrtia imprimantei sau dispozitivului de citit documente ş. a.). Informaţia pe suporturile în studiu este organizată în formă de **fişiere ex-**

terne în conformitate cu cerinţele sistemului de operare. Prin urmare, înainte de a fi utilizată, o variabilă fişier trebuie asociată cu un fişier extern. Metodele de asociere sînt specifice implementării limbajului şi sistemului de operare al calculatorului-gazdă.

În limbajul-standard asocierea se realizează prin includerea variabilelor de tip *fişier* ca argumente în antetul de program.

În Turbo PASCAL asocierea unei variabile de tip *fişier f* cu un fişier extern se realizează prin apelul de procedură

```
assign(f, s);
```

unde *s* este o expresie de tip **string** care specifică fişierul extern.

Exemple:

```
assign(FN, 'A:\REZULTAT\R.DAT')
```
– fişierul FN se asociază cu fişierul extern R.DAT din directorul REZULTAT de pe discul A:;

```
assign(FC, 'C:\A.CHR')
```
- fişierul FC se asociază cu fişierul A.CHR din directorul rădăcină al discului C:;

```
write('Daţi un nume de fişier:');
readln(str);
assign(FE, str)
```
- fişierul FE se asociază cu un fişier extern numele căruia este citit de la tastatură şi depus în variabila de tip **string** str.

După executarea instrucţiunii `assign(f, s)` toate operaţiile referitoare la fişierul PASCAL *f* se vor efectua asupra fişierului extern *s*.

Cele mai uzuale operaţii asupra unui fişier sînt citirea şi scrierea unei componente.

Citirea unei componente se realizează printr-un apel de forma

```
read(f, v),
```

unde *v* este o variabilă declarată cu tipul de bază al fişierului *f*.

Scrierea unei componente se realizează printr-un apel de forma:

```
write(f, e),
```

unde *e* este o expresie asociată cu tipul de bază al fişierului *f*.

Exemple:

```
read(FN, n);
write(FC, c);
read(FE, E);
```

După **tipul operaţiilor permise** asupra componentelor, fişierele se clasifică în:

– fişiere de intrare (este permisă numai citirea);

– fişiere de ieşire (este permisă numai scrierea);
– fişiere de actualizare (sînt permise scrierea şi citirea).

După **modul de acces** la componente, fişierele se clasifică în:

– fişiere cu acces secvenţial sau secvenţiale (accesul la componenta *i* este permis după ce s-a citit/scris componenta *i* – 1);
– fişiere cu acces aleator sau direct (orice componentă se poate referi direct prin numărul ei de ordine *i* în fişier).

Menţionăm că în limbajul-standard sînt permise numai fişiere secvenţiale de intrare sau ieşire.

Tipul fişierului (de intrare, ieşire sau de actualizare) şi modul de acces (secvenţial sau direct) se stabilesc printr-o operaţie de validare, numită **deschidere a fişierului**. Pentru aceasta, în limbajul-standard se utilizează următoarele proceduri:

`reset` *(f)* — pregăteşte un fişier existent pentru citire;

`rewrite` *(f)* — creează un fişier vid şi îl pregăteşte pentru scriere.

Cînd prelucrarea componentelor se termină, fişierul trebuie închis. La **închiderea** unui fişier sistemul de operare înscrie, dacă e necesar, elementul *EOF*; înregistrează fişierul extern nou creat în directorul respectiv ş.a.m.d.

În limbajul-standard se consideră că fişierele vor fi închise implicit la terminarea execuţiei programului respectiv. În Turbo PASCAL fişierul *f* ce închide prin apelul de procedură `close` *(f)*.

În concluzie, prezentăm ordinea în care trebuie apelate procedurile destinate prelucrării datelor de tip *fişier*:

1) `assign` *(f, s)* — asocierea *fişierului* PASCAL *f* cu fişierul extern *s*;

2) `reset` *(f)* /`rewrite` *(f)* — deschiderea fişierului *f* pentru citire/scriere;

3) `read` *(f, v)* /`write` *(f, e)* — citirea/scrierea unei componente a fişierului *f*;

4) `close` *(f)* — închiderea fişierului *f*.

După închiderea fişierului, variabila *f* poate fi asociată cu un alt fişier extern.

Întrucît valorile variabilelor de tip fişier se păstrează pe suporturile externe de informaţie, în PASCAL **atribuirile de fişiere sînt interzise**.

Întrebări şi exerciţii

❶ Explicaţi termenii *fişier PASCAL, fişier extern.*

❷ Unde se păstrează datele unui fişier PASCAL? Care este destinaţia procedurii `assign`?

❸ Reprezentaţi pe un desen structura următoarelor tipuri de date:

```
a) type  Tabel=array[ 1..5, 1..10] of real;
         FisierTabele =file of Tabel;
b) type  Multime=set of 'A'..'C';
         FisierMultimi=file of Multime;
c) type  Punct=record x, y:real end;
         Segment=record A, B:Punct end;
         FisierSegmente=file of Segment;
```

❹ Pentru ce sînt necesare operaţiile de deschidere şi închidere a fişierelor? Cum se realizează aceste operaţii?

❺ Explicaţi destinaţia procedurilor `read` şi `write`. Ce tip trebuie să aibă variabila *v* într-un apel de forma `read(`*f*`,` *v*`)`? Ce tip trebuie să aibă expresia *e* într-un apel de forma `write(`*f*`,` *e*`)`?

❻ Cum se clasifică fişierele în funcţie de operaţiile permise şi modul de acces?

❼ Variabilele `A` şi `B` sînt introduse prin declaraţia

```
var A, B : file of integer;
```

Este oare corectă instrucţiunea?

```
A:=B
```

Argumentaţi răspunsul.

4.7. FIŞIERE SECVENŢIALE

Fie definiţiile PASCAL

```
type  FT=file of T;
var   f : FT; v : T;
```

prin care au fost introduse tipul *fişier* `FT` cu tipul de bază `T`; variabila de tip *fişier f* şi variabila *v* de tipul `T`.

Pentru a deschide un **fişier secvenţial de ieşire** se apelează procedura `rewrite(`*f*`)`. În continuare în fişier se înscriu componentele respective. O componentă se înscrie printr-un apel de forma

`write(`*f*`,` *e*`)`,

unde *e* este o expresie de tipul `T`. O instrucţiune de forma

`write` $(f, e_1, e_2, ..., e_n)$

este echivalentă cu secvenţa de instrucţiuni

`write` (f, e_1); `write` (f, e_2); ...; `write` (f, e_n).

După înscrierea ultimei componente fişierul trebuie închis.

Exemplu:

```
Program P89;
  { Crearea unui fişier cu componente de tipul Elev }
  type Elev=record
```

```
              Nume : string;
              Prenume : string;
              NotaMedie : real
           end;
       FisierElevi=file of Elev;
var FE : FisierElevi;
    E : Elev;
    str : string;
    i, n : integer;
begin
write('Daţi numele fişierului de creat:  ');
readln(str);

assign(FE, str);
  {asociază FE cu numele din str }
rewrite(FE);
  {deschide FE pentru scriere }

write('Daţi numărul de elevi: '); readln(n);
for i:=1 to n do
  begin
    writeln('Daţi datele elevului  ', i);
      {citeşte cîmpurile variabilei E
    de la tastatură }
    write('Numele:  '); readln(E.Nume);
    write('Prenumele:  '); readln(E.Prenume);
    write('Nota medie:  '); readln(E.NotaMedie);
    write(FE, E); {scrie E în FE }
  end;
close(FE);          {închide FE }
readln;
end.
```

Pentru a **deschide un fişier secvenţial de intrare** se apelează procedura `reset` *(f)*. Componenta curentă se citeşte din fişier printr-un apel de forma:

`read`(f, v).

O instrucţiune de forma

`read`$(f, v_1, v_2, ..., v_n)$

este echivalentă cu secvenţa de instrucţiuni

`read`(f, v_1); `read`(f, v_2); ..., `read`(f, v_n).

Sfîrşitul de fişier este semnalizat de funcţia booleană `eof` *(f)* care ia valoarea `true` după citirea ultimei componente.

Exemplu:

```
Program P90;
  {Citirea unui fişier cu componente
    de tipul Elev }
 type Elev=record
                Nume : string;
                Prenume : string;
                NotaMedie : real
             end;
      FisierElevi=file of Elev;
 var FE : FisierElevi;
      E : Elev;
    str : string;
 begin
  write('Daţi numele fişierului de citit:  ');
  readln(str);

  assign(FE, str);   {asociază FE cu numele din str }
  reset(FE);         {deschide FE pentru citire }
  while not eof(FE) do
     begin
       read(FE, E);  {citeşte E din FE }
       writeln(E.Nume, ' ', E.Prenume,
               E.NotaMedie : 5:2);
                     {afişează E pe ecran }
     end;
  close(FE);         {închide FE }
  readln;
 end.
```

Subliniem faptul că numărul de componente ale unui fişier nu este cunoscut din declaraţia tipului respectiv. Teoretic, într-un fişier secvenţial de ieşire pot fi înscrise un număr infinit de componente. Practic, numărul componentelor este limitat de capacitatea de memorare a suportului extern de informaţie. Citirea consecutivă a componentelor unui fişier secvenţial de intrare se încheie cînd se ajunge la elementul *EOF*.

Întrebări şi exerciţii

❶ Cîte componente poate avea un fişier? În ce ordine se scriu/citesc componentele unui fişier secvenţial?

❷ Se consideră următoarele tipuri de date:

```
type Data=record
        Ziua : 1..31;
        Luna : 1..12;
        Anul : integer
     end;
     Persoana=record
        NumePrenume : string;
        DataNaşterii : Data
```

```
    end;
    FisierPersoane=file of Persoana;
```

Elaboraţi un program care citeşte de la tastatură datele referitoare la *n* persoane şi le înscrie într-un fişier.
Creaţi fişierele `FILE1.PRS`, `FILE2.PRS`, `FILE3.PRS` care trebuie să conţină datele referitoare, respectiv, la 2, 7 şi 10 persoane.

❸ Elaboraţi un program care citeşte fişiere create de programul din exerciţiul precedent şi afişează pe ecran:

a) toate persoanele din fişier;
b) persoanele născute în anul *a*;
c) persoanele născute pe data *z.l.a*;
d) persoana cea mai în vîrstă;
e) persoana cea mai tînără.

❹ Elaboraţi un program care afişează pe ecran media aritmetică a numerelor înscrise într-un fişier de tipul **file of** `real`.

❺ Într-un fişier de tipul **file of** `char` sînt înscrise caractere arbitrare. Elaboraţi un program care afişează pe ecran numărul vocalelor din fişier.

❻ Comentaţi următorul program:

```
Program P91;
  { Eroare }
 type FisierNumere=file of integer;
 var FN : FisierNumere;
      i : integer;
      r : real;
      s : string;
 begin
  Writeln('Daţi numele fişierului de creat:  ');
  readln(s);
  assign(FN, s);
  rewrite(FN);
  i:=1;
  write(FN, i);
  i:=10;
  write(FN, i);
  r:=20;
  write(FN, r);
  close(FN);
 end.
```

4.8. FIŞIERE CU ACCES DIRECT

Acest paragraf se referă în întregime la implementarea Turbo PASCAL.

În Turbo PASCAL variabilele de tip **file of** ***T*** permit atît citiri, cît şi scrieri, indiferent dacă ele au fost deschise cu `rewrite` sau

`reset`. Pentru aceasta fiecare componentă a fişierului are asociat un număr, numit *numărul componentei*. Prima componentă are numărul 0. Cea de-a doua componentă are numărul 1 ş.a.m.d. Numărul de componente ale unui fişier *f* poate fi determinat cu ajutorul funcţiei standard `FileSize(`*f*`)`.

În mod normal, accesul la componentele fişierului este secvenţial. Aceasta înseamnă că după citirea/scrierea unei componente, poziţia curentă de fişier se deplasează la următoarea componentă, în sensul ordonării numerice. Accesul direct se bazează pe procedura de căutare `Seek(`*f*`, i)`, care mută poziţia curentă în fişier pe componenta `i`. După această poziţionare componenta astfel aleasă poate fi citită/scrisă. Funcţia-standard `FilePos(`*f*`, i)` permite determinarea poziţiei curente în fişier.

Exemplu:

```
Program P92;
  {Accesul direct la fişiere de tipul FisierElev }
 type Elev=record
              Nume : string;
              Prenume : string;
              NotaMedie : real
           end;
      FisierElevi=file of Elev;
 var FE : FisierElevi;
      E : Elev;
    str : string;
   i, n : integer;
      r : char;
 begin
  write('Daţi numele fişierului:  ');
  readln(str);
  assign(FE, str);
  reset(FE);
  n:=FileSize(FE);
  writeln('Fisierul', str, 'are', n, 'componente');
  writeln('Numeraţia componentelor: 0..', n-1);
  write('Daţi numărul componentei:  '); readln(i);
  if i>n-1 then
     writeln('Eroare: componentă inexistentă')
    else
      begin
        Seek(FE, i);
          {poziţionare pe componenta i }
        read(FE, E);
          {citirea componentei selectate }
        writeln(E.Nume, ' ', E.Prenume,
                E.NotaMedie :5:2);
        write('Modificăm componenta? [d/n]');
```

```
        readln(r);
        if r='d' then
          begin
            write('Numele:  '); readln(E.Nume);
            write('Prenumele:  ');
            readln(E.Prenume);
            write('Nota medie:  ');
            readln(E.NotaMedie);
            Seek(FE, i);
             {poziţionare pe componenta i }
            write(FE, E);
             { scrierea componentei modificate }
          end;
      end;
  close(FE);
  readln;
 end.
```

Amintim că fişierele de tipul `FisierElevi` pot fi create cu ajutorul programului `P89` din paragraful precedent.

Întrebări şi exerciţii

❶ Prin ce se deosebeşte accesul direct de cel secvenţial?

❷ Care este destinaţia funcţiilor şi procedurii `FileSize`, `FilePos`, `Seek`?

❸ Creaţi cu ajutorul programului `P89` fişierele `CLASA0.ELV`, `CLASA1.ELV` şi `CLASA5.ELV`. Fişierele trebuie să conţină, respectiv, 0, 1 şi 5 componente. Verificaţi funcţionarea programului `P92` pentru fiecare din fişierele create.

❹ Elaboraţi un program care asigură accesul direct la componentele fişierelor de tipul `FisierPersoane` (vezi exerciţiul 2 din paragraful precedent). Programul va afişa la ecran şi va modifica, dacă se va cere, componenta indicată de utilizator.

❺ Se consideră următoarele tipuri de date:

```
type Angajat= record
                NumePrenume : string;
                ZileLucrate : 1..31;
                PlataPeZi : real;
                PlataPeLuna : real
              end;
FisierAngajati=file of Angajat;
```

Elaboraţi un program, care:

a) citeşte de la tastatură datele referitoare la fiecare angajat şi creează fişiere de tipul `FisierAngajati`;
b) modifică, dacă e necesar, cîmpurile `ZileLucrate`, `PlataPeZi` şi `PlataPeLuna` ale componentelor indicate din fişierul respectiv;
c) calculează salariul mediu al angajaţilor incluşi în fişier;
d) afişează la ecran datele despre angajaţii cu plata lunară maximală;
e) ordonează angajaţii din fişier în ordine alfabetică;
f) ordonează angajaţii din fişier în ordinea creşterii plăţilor pe lună;

g) adaugă noi angajaţi în fişierul existent;
h) exclude din fişier persoanele concediate.

4.9. FIŞIERE *TEXT*

E cunoscut faptul că datele fişierelor PASCAL se păstrează pe suporturile externe de informaţie. În cazul fişierelor definite prin declaraţii de forma `file of` *T* componentele de tip *T* sînt reprezentate pe suporturile respective în **forma internă**, şi anume, prin secvenţe de cifre binare. Acest mod de reprezentare a datelor este convenabil în cazul memoriilor externe (discurile şi benzile magnetice, discurile optice ş. a.). În cazul echipamentelor de intrare-ieşire (tastatura, ecranul, imprimanta etc.) datele respective trebuie reprezentate în **forma externă**, şi anume, prin secvenţe de caractere.

Pentru a facilita interacţiunea între om şi sistemul de calcul, în PASCAL informaţia destinată utilizatorului se reprezintă în formă de fişiere *text*. Un fişier *text* este format dintr-o secvenţă de caractere divizată în linii (*fig. 4.9*). Lungimea liniilor este variabilă. Sfîrşitul fiecărei linii este indicat de un element special, notat *EOL* (*End Of Line* — sfîrşit de linie). Întrucît lungimea liniilor este variabilă, poziţia unei linii în cadrul fişierului nu poate fi calculată din timp. În consecinţă, accesul la componentele fişierelor *text* este **secvenţial**.

Un fişier *text* se defineşte printr-o declaraţie de forma

```
var f : text;
```

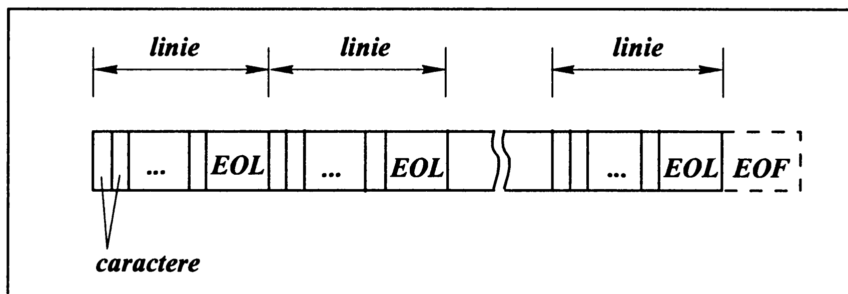


Fig. 4.9. Structura unui fişier *text*

tipul predefinit `text` fiind cunoscut oricărui program PASCAL. Subliniem faptul că tipurile `text` şi `file of char` sînt distincte, întrucît fişierul `file of char` nu include elemente *EOL* (*fig. 4.8*).

Fişierele *text* pot fi prelucrate cu ajutorul procedurilor cunoscute, aplicabile oricărui tip de fişiere: `assign`, `reset`, `rewrite`, `read`,

`write`, `close`. În completare, limbajul include proceduri speciale, destinate prelucrării elementelor *EOL*:

`writeln` *(f)* — înscrie în fişier elementul *EOL* (sfîrşit de linie);

`readln` *(f)* — trece la linia următoare.

Sfîrşitul de linie este semnalat de funcţia booleană `eoln` *(f)* care ia valoarea `true` după citirea ultimului caracter din linie.

O instrucţiune de forma

`writeln` $(f, e_1, e_2, .., e_n)$

este echivalentă cu secvenţa de instrucţiuni

`write` $(f, e_1, e_2, .., e_n)$; *writeln(f)* .

O instrucţiune de forma

`readln` $(f, v_1, v_2, .., v_n)$

este echivalentă cu secvenţa de instrucţiuni

`read` $(f, v_1, v_2, .., v_n)$; *readln(f)* .

Pentru introducerea şi extragerea datelor, de regulă, se utilizează fişierele *text* predefinite `Input` şi `Output`. Fişierul `Input` este destinat numai pentru operaţii de citire şi este asociat cu fişierul de intrare al sistemului de operare (de regulă tastatura). Fişierul `Output` este destinat numai pentru operaţii de scriere şi este asociat cu fişierul-standard de ieşire al sistemului de operare (de regulă ecranul). Aceste fişiere sînt deschise şi închise automat la începutul şi, respectiv, la sfîrşitul execuţiei programului. Dacă numele fişierului nu este specificat în lista parametrilor unui apel de subalgoritm, se presupune că fişierul *text* implicit este `Input` sau `Output`, în funcţie de natura subalgoritmului. De exemplu, `read(c)` este echivalent cu `read(Input,c)`, iar `write(c)` este echivalent cu `write(Output, c)`.

Pentru exemplificare prezentăm programul `P93` care creează pe discul curent fişierul text `FILE.TXT`. Liniile fişierului sînt introduse de la tastatură (fişierul `Input`). Sfîrşitul liniei se indică prin acţionarea tastei <*ENTER*>, iar sfîrşitul fişierului prin acţionarea tastelor <*CTRL+Z*>, <*ENTER*>.

```
Program P93;
  { Crearea fişierului text FILE.TXT }
 var F : text;
     c : char;
  begin
  assign(F, 'FILE.TXT'); { asociază F cu FILE.TXT }
  rewrite(F);            { deschide F pentru scriere }
```

```
  while not eof do          { eof(Input) }
    begin
      while not eoln do     { eoln(Input) }
        begin
          read(c);          {  citeşte c din Input }
          write(F, c);      {  scrie c în F }
         end;
      writeln(F);           { scrie EOL în F }
      readln;               { trece la linia
                             următoare din Input }
    end;
  close(F);                 { închide F }
 end.
```

Programul ce urmează afişează conţinutul fişierului `FILE.TXT` pe ecran.

```
Program P94;
  {Citirea fişierului text FILE.TXT }
 var F : text;
     c : char;
 begin
  assign(F, 'FILE.TXT');{ asociază F cu FILE.TXT }
  reset(F);              { deschide F pentru citire }
  while not eof(F) do
    begin
      while not eoln(F) do
        begin
          read(F, c);    {citeşte c din F }
          write(c);      { scrie c în Output }
        end;
      readln(F);   { trece la linia următoare din F }
      writeln;           { scrie EOL în Output }
    end;
  close(F);              { închide F }
  readln;
 end.
```

Explorarea caracter cu caracter a fişierelor *text* este greoaie atunci cînd secvenţele de caractere din text trebuie interpretate ca formînd date de tip `integer`, `real`, `boolean`, *şir de caractere*. Conversia între forma externă şi reprezentarea internă a acestor tipuri ar cădea în sarcina programatorului. În consecinţă, procedurile de citire/înscriere sînt extinse în felul următor.

În cazul fişierelor *text* variabila `v` dintr-un apel `read`*(f, v)* poate fi de tipul `integer`, `real`, `char` sau *şir de caractere*. La citire secvenţa de caractere care reprezintă variabila `v` va fi transformată în reprezentarea internă.

Expresia *e* dintr-un apel `write` *(f, e)* poate fi urmată de specificatori de format. Valoarea expresiei poate fi de tipul `integer`, `real`, `boolean`, `char` sau *şir de caractere*. La scriere valoarea respectivă este transformată din reprezentarea internă într-o secvenţă de caractere.

Secvenţele de caractere citite/scrise de procedurile `read/write` se conformează sintaxei constantelor de tipul variabilei/expresiei *v/e*.

Utilizarea procedurilor `write`, `writeln`, `read` şi `readln` pentru prelucrarea fişierelor `Output` şi `Input` a fost studiată în detalii în paragrafele 3.7 şi 3.8. La fel se prelucrează orice fişier *text* definit de utilizator.

Pentru exemplificare prezentăm programul `P95` care citeşte de la tastatură cîte trei numere reale *a, b, c* pe care le scrie în fişierul `IN.TXT`. Apoi citind aceste trei numere, reprezentînd laturile unui triunghi, scrie în fişierul `OUT.TXT` numerele *a, b* şi *c*, semiperimetrul *p* şi aria triunghiului *s*. În continuare, conţinutul fişierului `OUT.TXT` este afişat pe ecran.

```
Program P95;
  { Prelucrarea fişierelor IN.TXT şi OUT.TXT }
 var  F, G : text;
      a, b, c, p, s : real;
      str : string;
 begin
  assign(F, 'IN.TXT'); { asociază F cu IN.TXT }
  rewrite(F);          { deschide F pentru scriere }
  writeln('Daţi numerele reale a, b, c:');
  while not eof do
    begin
      readln(a, b, c); { citeşte a,b,c de la tastatură}
      writeln(F, a:8:2, b:8:2, c:8:2);
                       { scrie a,b,c, în F }
    end;
  close(F);            { închide F }
  reset(F);            { deschide F
                        pentru citire }
  assign(G, 'OUT.TXT'); { asociază G cu OUT.TXT }
  rewrite(G);          { deschide G pentru scriere }
  while not eof(F) do
    begin
      readln(F, a, b, c);  { citeşte a,b,c din F }
      write(G, a:8:2, b:8:2, c:8:2);
                           { scrie a,b,c în G }
      p:=(a+b+c)/2;
      s:=sqrt(p*(p-a)*(p-b)*(p-c));
      writeln(G, p:15:2, s:15:4);
                       { scrie p, s în G }
    end;
```

```
  close(F);                { închide F }
  close(G);                { închide G }
  reset(G);                { deschide G pentru citire }
  while not eof(G) do
    begin
      readln(G, str);      { citeşte str din G }
      writeln(str);        { afişează str pe ecran }
    end;
  close(G);                { închide G }
  readln;
end.
```

Pentru datele de intrare

1 1 1 *<ENTER>*
3 4 6 <ENTER>
<CTRL+Z><ENTER>

programul P95 afişează pe ecran:

```
1.00 1.00 1.00 1.50 0.4330
3.00 4.00 6.00 6.50 5.3327
```

Întrebări şi exerciţii

❶ Care este diferenţa dintre un fişier *text* şi un fişier **file of** char?
❷ Explicaţi semnificaţia elementelor *EOL* şi *EOF*.
❸ Care este diferenţa dintre procedurile read şi readln? Procedurile write şi writeln?
❹ Lansaţi în execuţie următorul program:

```
Program P96;
  { Asocierea fişierului FN cu consola }
 type FisierNumere=file of integer;
 var FN : FisierNumere;
      i : integer;
 begin
  assign(FN, 'CON');
  rewrite(FN);
  i:=1;
  write(FN, i);
  i:=2;
  write(FN, i);
  i:=3;
  write(FN, i);
  close(FN);
  readln;
 end.
```

Explicaţi rezultatele afişate pe ecran.
❺ Elaboraţi un program care afişează pe ecran conţinutul oricărui fişier *text*.

❻ Elaboraţi un program care afişează pe ecran numărul de vocale dintr-un fişier *text*.

❼ Datele de intrare ale unui program sînt înmagazinate într-un fişier *text*. Fiecare linie a fişierului conţine două numere întregi şi trei numere reale separate prin spaţii. Elaboraţi un program care afişează suma numerelor întregi şi suma numerelor reale din fiecare linie pe ecran.

❽ Datele de intrare ale unui program sînt înmagazinate într-un fişier *text*. Fiecare linie a fişierului conţine trei numere reale separate prin spaţii şi unul din cuvintele `ADMIS`, `RESPINS`. Elaboraţi un program care:

a) afişează conţinutul fişierului în studiu pe ecran;

b) creează o copie de rezervă a fişierului;

c) creează un fişier *text* liniile căruia conţin media celor trei numere reale din liniile respective ale fişierului de intrare;

d) tipăreşte la imprimantă liniile fişierului de intrare, precedate de numerele de ordine 1, 2, 3 ş. a. m. d.

❾ Fiecare linie a fişierului *text* conţine următoarele date, separate prin spaţii:

numărul de ordine (`integer`);
numele (un `string` ce nu conţine spaţii);
nota la disciplina 1 (`real`);
nota la disciplina 2 (`real`);
nota la disciplina 3 (`real`).

Elaboraţi un program care:

a) creează o copie de rezervă a fişierului în studiu;

b) tipăreşte conţinutul fişierului la imprimantă;

c) creează un fişier liniile căruia conţin următoarele date separate prin spaţii:

numărul de ordine (`integer`);
numele (`string`);
nota medie (`real`).

Fişierul creat în punctul *c* trebuie afişat pe ecran şi tipărit la imprimantă.

Capitolul 5

FUNCŢII ŞI PROCEDURI

5.1. SUBPROGRAME

E cunoscut faptul că o problemă complexă poate fi rezolvată prin divizarea ei într-un set de părţi mai mici (subprobleme). Pentru fiecare parte se scrie o anumită secvenţă de instrucţiuni, denumită **subprogram**. Problema în ansamblu se rezolvă cu ajutorul programului principal, în care pentru rezolvarea subproblemelor se folosesc apelurile subprogramelor respective. Cînd în programul principal se întîlneşte un apel, execuţia continuă cu prima instrucţiune din programul apelat (*fig. 5.1*). Cînd se termină executarea instrucţiunilor din subprogram, se revine la instrucţiunea imediat următoare apelului din programul principal.

În limbajul PASCAL există două tipuri de subprograme, şi anume, funcţii şi proceduri:

<*Subprograme*> ::= { <*Funcţie*>; | <*Procedură*>; }

Funcţiile sînt subprograme care calculează şi returnează o valoare. Limbajul PASCAL conţine un set de funcţii predefinite, cunoscute oricărui program: `sin`, `cos`, `eof` etc. (vezi *tabelul 3.1*). În completare, programatorul poate defini funcţii proprii, care se apelează în acelaşi mod ca funcţiile-standard. Prin urmare, conceptul de funcţie extinde noţiunea de **expresie** PASCAL.

Procedurile sînt subprograme care efectuează prelucrarea datelor comunicate în momentul apelului. Limbajul conţine procedurile predefinite `read`, `readln`, `write`, `writeln` ş.a., studiate în capitolele precedente. În completare, programatorul poate defini proceduri proprii, care se apelează în acelaşi mod ca procedurile-standard. Prin urmare, conceptul de procedură extinde noţiunea de **instrucţiune** PASCAL.

Subprogramele se definesc, în întregime, în partea declarativă a unui program (*fig. 3.14*). Evident, apelurile de funcţii şi proceduri se includ în partea executabilă a programului.

Un subprogram poate fi apelat chiar de el însuşi, caz în care apelul este **recursiv**.

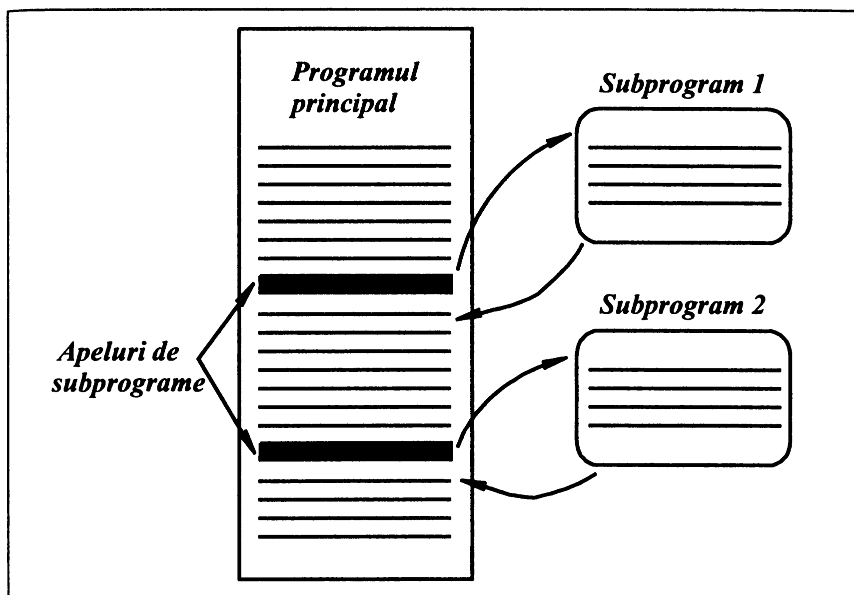


Fig. 5.1. Interacţiunea între program şi subprogram

Întrebări şi exerciţii

❶ Explicaţi termenii *program principal* şi *subprogram.*
❷ Cum interacţionează programul şi subprogramul?
❸ Care este diferenţa dintre proceduri şi funcţii?
❹ Cum se apelează o funcţie? În care instrucţiuni ale limbajului pot apărea apeluri de funcţii?
❺ Cum se apelează o procedură?
❻ Numiţi tipul argumentului şi tipul rezultatului furnizat de funcţiile predefinite `abs, chr, eof, eoln, exp, ord, sin, sqr, sqrt, pred, succ, trunc`.
❼ Numiţi tipul parametrilor actuali ai procedurilor `read` şi `write`.
❽ Ce prelucrări de date efectuează procedurile `read` şi `write`?

5.2. FUNCŢII

Textul PASCAL al unei **declaraţii de funcţie** are forma:

```
function f (x1; x2 ...; xn): tr ;
D;
begin
  ...
  f:=e;
  ...
end;
```

Prima linie este antetul funcţiei, format din:

f — numele funcţiei;

$(x_1;\ x_2\ ...;\ x_n)$ — lista opţională de parametri formali reprezentînd argumentele funcţiei;

t_r — tipul rezultatului; acesta trebuie să fie numele unui tip simplu sau tip referinţă.

Antetul este urmat de **corpul funcţiei** format din declaraţiile locale opţionale D şi instrucţiunea compusă **begin** ... **end**.

Declaraţiile locale sînt grupate în secţiunile (eventual vide) **label**, **const**, **type**, **var**, **function/procedure**.

Numele f al funcţiei apare cel puţin o dată în partea stîngă a unei instrucţiuni de atribuire care se execută: f:=e. Ultima valoare atribuită lui f va fi întoarsă în programul principal.

În mod obişnuit, un parametru formal din lista $(x_1;\ x_2\ ...;\ x_n)$ are forma:

$$(v_1,\ v_2\ ...,\ v_k):\ t_p$$

unde $(v_1,\ v_2\ ...,\ v_k)$ sînt identificatori, iar t_p este un nume de tip.

Utilizarea funcţiei f se specifică printr-un apel de forma:

$$f(a_1,\ a_2\ ...,\ a_n)$$

unde $(a_1,\ a_2\ ...,\ a_n)$ este **lista de parametri actuali**. De obicei, parametrii actuali sînt expresii, valorile cărora sînt comunicate funcţiei. Corespondenţa între un parametru actual şi parametrul formal se face prin poziţia ocupată de aceştia în cele două liste. Parametrul actual trebuie să fie **compatibil** din punct de vedere al atribuirii cu tipul parametrului formal.

Exemplu:

```
Program P97;
  { Declararea şi utilizarea funcţiei Putere }
  type Natural=0..MaxInt;
  var a : real;
      b : Natural;
      c : real;
      s : integer;
      t : integer;
      v : real;

  function Putere(x : real; n : Natural) : real;
   { calcularea lui x la puterea n }
  var p : real;
      i : integer;
  begin
    p:=1;
  for i:=1 to n do p:=p*x;
    Putere:=p;
```

```
  end; { Putere }

  begin
    a:=3.0;
    b:=2;
    c:=Putere(a, b);
    writeln(a:10:5, b:4, c:10:5);

    s:=2;
    t:=4;
    v:=Putere(s, t);
    writeln(s:5, t:4, v:10:5);

    readln;
  end.
```

Funcţia `Putere` are doi parametri formali: `x` de tipul `real` şi `n` de tipul `Natural`. Funcţia returnează o valoare de tipul `real`. În corpul funcţiei sînt declarate variabilele locale `p` şi `i`.

La execuţia apelului `Putere(a, b)` valorile `3.0` şi `2` ale parametrilor actuali `a`, `b` se transmit parametrilor formali, respectiv, `x` şi `n`. De menţionat că tipul lui `a` coincide cu tipul lui `x` şi tipul lui `b` coincide cu tipul lui `n`.

În cazul apelului `Putere(s, t)` tipul parametrilor actuali `s`, `t` nu coincide cu tipul parametrilor formali, respectiv, `x` şi `n`. Totuşi, apelul este corect, întrucît tipurile respectve sînt compatibile din punct de vedere al atribuirii.

Întrebări şi exerciţii

❶ Se consideră următoarea declaraţie:

```
function Factorial(n : integer) : integer;
var p, i : integer;
begin
  p:=1;
  for i:=1 to n do p:=p * i;
  Factorial:=p;
end;
```

Numiţi tipul parametrului formal şi tipul rezultatului returnat de funcţie. Precizaţi variabilele declarate în corpul funcţiei.

Elaboraţi un program care afişează pe ecran valorile *n*! pentru *n* = 2, 3 şi 7.

❷ În care loc al programului principal se includ declaraţiile de funcţii?

❸ Comentaţi programul ce urmează:

```
Program P98;
 { Eroare }
function Factorial(n : 0..7) : integer;
var p, i : integer;
```

```
begin
 p:=1;
 for i:=1 to n do p:=p*i;
 Factorial:=p;
end; { Factorial }

begin
 writeln(Factorial(4));
 readln;
end.
```

❹ Se consideră antetul

```
function F(x : real; y : integer;
                              z : char) : boolean;
```

Care din apelurile ce urmează sînt corecte:

a) F(3.18, 4, *'a'*);
b) F(4, 4, *'4'*);
c) F(4, 4, 4);
d) F(4, 3.18, *'a'*);
e) F(3.18, 4, 4);
f) F(*'3.18'*, 4, *'4'*);
g) F(15, 21, *'3'*);
h) F(15,21,3).

❺ Elaboraţi o funcţie care calculează:

a) suma numerelor reale *a, b, c, d*;
b) media numerelor întregi *i, j, k, m*;
c) minimumul din numerele *a, b, c, d*;
d) numărul de vocale într-un şir de caractere;
e) numărul de consoane într-un şir de caractere;
f) rădăcina ecuaţiei $ax + b = 0$;
g) cel mai mic divizor al numărului întreg $n > 0$, diferit de 1;
h) cel mai mare divizor comun al numerelor naturale *a, b*;
i) cel mai mic multiplu comun al numerelor naturale *a, b*;
j) ultima cifră în notaţia zecimală a numărului întreg $n > 0$;
k) cîte cifre sînt în notaţia zecimală a numărului întreg $n > 0$;
l) cifra superioară în notaţia zecimală a numărului întreg $n > 0$;
m) numărul de apariţii a caracterului dat într-un şir de caractere.

❻ Se consideră următoarele declaraţii:

```
const nmax=100;
type Vector=array [1..nmax] of real;
```

Elaboraţi o funcţie care calculează :

a) suma componentelor unui vector;
b) media componentelor vectorului;
c) componenta maximă;
d) componenta minimă.

❼ Se consideră următoarele tipuri de date:

```
type Punct=     record
                  x, y : real
                end;
    Segment=    record
                  A, B : Punct
                end;
    Triunghi=   record
                  A, B, C : Punct
                end;
    Dreptunghi=record
                  A, B, C, D : Punct
                end;
    Cerc=       record
                  Centru : Punct;
                  Raza : real
                end;
```

Elaboraţi o funcţie care calculează:

a) lungimea segmentului;
b) lungimea cercului;
c) aria cercului;
d) aria triunghiului;
e) aria dreptunghiului.

❽ Variabila *A* este introdusă prin declaraţia

```
var A : set of char;
```

Elaboraţi o funcţie care returnează numărul de caractere din mulţimea *A*.

❾ Elaboraţi o funcţie care să calculeze diferenţa în secunde între două momente de timp date prin oră, minute şi secunde.

❿ Un triunghi este definit prin coordonatele vîrfurilor sale. Scrieţi funcţii care, pentru două triunghiuri date, să studieze dacă:

a) au aceeaşi arie;
b) sînt asemenea;
c) primul este în interiorul celui de-al doilea.

5.3. PROCEDURI

Forma generală a textului unei declaraţii de procedură este:

```
procedure p (x1; x2...; xn);
D;
begin
  ...
end;
```

În **antetul procedurii** apar:

p — numele procedurii;
$(x_1;\ x_2\ ...;\ x_n)$ — lista opţională de parametri formali;

În **corpul procedurii** sînt incluse:

D — declaraţiile locale (opţionale) grupate după aceleaşi reguli ca în cazul funcţiilor;
begin ... **end** — instrucţiune compusă; ea nu conţine vreo atribuire asupra numelui procedurii.

Procedura poate să întoarcă mai multe rezultate, dar nu prin numele ei, ci prin variabile desemnate special (cu prefixul **var**) în lista de parametri formali.

Parametrii din listă introduşi prin declaraţii de forma

$v_1,\ v_2\ ...,\ v_k\colon\ t_p$

se numesc ***parametri-valoare***. Aceştia servesc pentru transmiterea de valori din programul principal în procedură.

Parametrii formali introduşi în listă prin declaraţii de forma

var $v_1,\ v_2\ ...,\ v_k\colon\ t_p$

se numesc ***parametri-variabilă*** şi servesc pentru întoarcerea rezultatelor din procedură în programul principal.

Activarea unei proceduri se face printr-un apel de forma

$p(a_1,\ ...,\ a_n)$

unde $(a_1,\ ...,\ a_n)$ este lista de ***parametri actuali***. Spre deosebire de funcţie, apelul de procedură este o instrucţiune; aceasta se inserează în programul principal în locul în care sînt dorite efectele produse de execuţia procedurii.

În cazul unui ***parametru-valoare*** drept parametru actual poate fi utilizată orice expresie de tipul respectiv, în particular o constantă sau o variabilă. Modificările parametrilor valoare nu se transmit în exteriorul subprogramului.

În cazul unui ***parametru-variabilă*** drept parametri actuali pot fi utilizate numai variabile. Evident, modificările parametrilor în studiu vor fi transmise programului apelant.

Exemplu:

```
Program P99;
  { Declararea şi utilizarea procedurii Lac }
var a, b, c,
    t, q : real;

procedure Lac(r : real; var l, s : real);
  { lungimea şi aria cercului       }
  { r - raza; l - lungimea; s - aria }
```

```
const Pi=3.14159;
begin
   l:=2*Pi*r;
   s:=Pi*sqr(r);
end; { Lac }

begin
   a:=1.0;
   Lac(a, b, c);
   writeln(a:10:5, b:10:5, c:10:5);

   Lac(3.0, t, q);
   writeln(3.0:10:5, t:10:5, q:10:5);

   readln;
end.
```

Procedura `Lac` are trei parametri formali: `r`, `l` şi `s`. Parametrul `r` este un parametru-valoare, iar `l` şi `s` sînt parametri-variabilă.

Execuţia instrucţiunii `Lac(a,b,c)` determină transmiterea valorii `1.0` drept valoare a parametrului formal `r` şi a locaţiilor (adreselor) variabilelor `b` şi `c` drept locaţii (adrese) ale parametrilor formali `l` şi `s`. Prin urmare, secvenţa de instrucţiuni

```
a:=1.0;
Lac(a,b,c)
```

este echivalentă cu secvenţa

```
b:=2*Pi*1.0;
c:=Pi*sqr(1.0).
```

În mod similar, instrucţiunea

```
Lac(3.0,t,q)
```

este echivalentă cu secvenţa

```
t:=2*Pi*3.0;
q:=Pi*sqr(3.0).
```

Întrebări şi exerciţii

❶ Care este diferenţa dintre un *parametru-valoare* şi un *parametru-variabilă*?
❷ Se consideră declaraţiile:

```
Var k, m, n : integer;
    a, b, c : real;
procedure P(i : integer; var j : integer;
            x : real; var y : real);
begin
  {...}
end.
```

Care din apelurile ce urmează sînt corecte?

a) `P(k,m,a,b);` f) `P(n,m,6,b);`
b) `P(3,m,a,b);` g) `P(n,m,6,20);`
c) `P(k,3,a,b);` h) `P(a,m,b,c);`
d) `P(m,m,a,b);` i) `P(i,i,i,i);`
e) `P(m,k,6.1,b);` j) `P(a,a,a,a).`

Argumentaţi răspunsul.

❸ Comentaţi programul ce urmează:

```
Program P100;
 { Eroare }
var a : real;
    b : integer;
procedure P(x : real; var y : integer);
begin
 { ... }
end; {  P }
begin
 P(a, b);
 P(0.1, a);
 P(1, b);
 P(a, 1);
end.
```

❹ Ce va afişa pe ecran programul ce urmează?

```
Program P101;
  { Parametru-valoare şi parametru-variabilă }
var a, b : integer;

procedure P(x : integer; var y : integer);
begin
 x:=x+1;
 y:=y+1;
 writeln('x=', x, ' y=', y);
end; { P }

begin
 a:=0;
 b:=0;
 P(a, b);
 writeln('a=', a, ' b=', b);
 readln;
end.
```

Argumentaţi răspunsul.

❺ Elaboraţi o procedură care:

a) calculează rădăcinile ecuaţiei $ax^2 + bx + c = 0$;
b) radiază dintr-un şir caracterul indicat în apel;
c) încadrează un şir de caractere între simbolurile "#";
d) ordonează componentele unui tablou **array**[1..100] **of** real în ordine crescătoare;
e) ordonează componentele unui fişier **file of** integer în ordine descrescătoare;
f) calculează şi depune într-un tablou numerele prime mai mici decît un număr natural dat *n*.

❻ În exerciţiul 4 din paragraful 4.3 este definit tipul de date `ListaPersoane`. Elaboraţi procedurile necesare pentru calcularea datelor specificate în punctele a, b, c, ..., k ale exerciţiului în studiu.

❼ Elaboraţi o procedură care:

a) creează o copie de rezervă a unui fişier *text*;
b) exclude dintr-un fişier *text* liniile vide;
c) numerotează liniile unui fişier *text*;
d) concatenează două fişiere *text* într-unul singur;
e) concatenează *n* fişiere *text* (*n*>2) într-unul singur.

❽ În exerciţiul 5 din paragraful 4.9 este definit tipul de date `FisierAngajati`. Elaboraţi procedurile necesare pentru prelucrarea datelor conform specificărilor din punctele a, b, c, ..., h ale exerciţiului în studiu.

❾ Să se definească un tip de date pentru numere naturale foarte mari şi să se scrie proceduri care să adune şi să scadă astfel de numere.

5.4. PARAMETRI FORMALI *FUNCŢIE / PROCEDURĂ*

În completare la parametrii-valoare şi parametrii-variabilă, limbajul PASCAL permite utilizarea ***parametrilor-funcţie*** şi ***parametrilor-procedură***.

Specificarea unui parametru formal *funcţie/procedură* are forma antetului acestei funcţii/proceduri. De exemplu, în declaraţia

```
procedure Q(function F(x:real):real,
            i : integer);
begin
  {...}
end;
```

apare ca parametru-*funcţie* antetul **function** F(x:real):real.

Un apel al procedurii Q ar putea fi

```
Q(F1, j),
```

unde F1 este numele unei funcţii cunoscute în locul apelului.

În Turbo PASCAL parametrii-*funcţie/procedură* se declară explicit ca aparţinînd unui tip procedural.

Tipul procedural se defineşte cu ajutorul cuvîntului-cheie **type**. Definiţia respectivă specifică numele şi tipul parametrilor formali ai unui subprogram. În cazul funcţiilor se indică şi tipul valorii returnate.

Exemplu:

```
type TF=function(x : real) : real;
     TP=procedure(i, j : integer);
var  v : TF;
     p : TP;
```

Mulţimea valorilor tipului `TF` este formată din denumirile tuturor funcţiilor PASCAL de o variabilă reală cu rezultate reale. Mulţimea valorilor tipului `TP` este formată din denumirile tuturor procedurilor PASCAL de două variabile întregi.

Menţionăm că la definirea tipurilor procedurale nu apare nici un fel de nume de funcţie/procedură.

Asupra variabilelor de tip procedural pot fi efectuate atribuiri. De exemplu, în prezenţa declaraţiilor

```
function F1(x : real) : real;
begin
  {...}
end;

procedure P1(m, n : integer);
begin
  {...}
end;
```

atribuirile

```
v:=F1;
p:=P1
```

sînt corecte.

Folosirea tipurilor procedurale simplifică declararea parametrilor formali *funcţie/procedură*. Un astfel de parametru se specifică acum ca un caz particular de parametru-valoare printr-un **nume** urmat de un **tip procedural**. Parametrul actual corespunzător va fi un nume de *funcţie/procedură* de tip compatibil celui procedural.

Exemplu:

```
Program P102;
 { Parametru formal funcţie }

type TF=function(x : real) : real;
var v : TF;

function F1(x : real) : real;
```

```
begin
 F1:=x
end; { F1 }

function F2(x : real) : real;
begin
  F2:=3*x+2
end; { F2 }

function F3(x : real) : real;
begin
 F3:=sqr(x)+2*x+1
end; { F3 }

procedure Tabel(F : TF; x1, x2, dx : real);
{ Tabelarea funcţiei F(x) }
var a, b : real;
begin
 a:=x1;
 while a<=x2 do
    begin
      b:=F(a);
      writeln(a:15:5, b:15:5);
      a:=a+dx;
    end;
end; { Tabel }

begin
 writeln('Tabelul funcţiei F1');
 Tabel(F1, 0, 2, 0.5);

 writeln('Tabelul funcţiei F2');
 Tabel(F2, 0, 2, 0.5);

 writeln('Tabelul funcţiei F3');
 v:=F3;
 Tabel(v, 0, 2, 0.5);
 readln;
end.
```

În programul în studiu sînt definite:

— tipul procedural `TF`;
— variabila `v` de tipul `TF`;
— funcţiile `F1`, `F2` şi `F3` de tipul `TF`;
— procedura `Tabel` cu parametrul-funcţie `F` şi parametrii-valoare `x1`, `x2` şi `dx`.

Procedura `Tabel` afişează pe ecran tabelul oricărei funcţii de tipul `TF`. În corpul programului această procedură este apelată pentru tabe-

larea funcţiilor `F1`, `F2` şi `F3`. În ultimul apel numele funcţiei `F3` se transmite procedurii `Tabel` prin intermediul variabilei `v`.

Subliniem faptul că funcţiile şi procedurile predefinite nu pot fi transmise ca parametri actuali. De exemplu, apelul procedurii `Tabel (sin, ...)` nu este corect, deoarece `sin` este o funcţie predefinită. Inconvenientul poate fi însă depăşit definind o funcţie `fsin`

```
function fsin(x : real) : real;
begin
  fsin:=sin(x);
end; { fsin}
```

care poate fi folosită în apelul `Tabel (fsin, ...)` cu efectul dorit.

Întrebări şi exerciţii

❶ Cum se specifică un parametru formal *funcţie/procedură* în PASCAL? În Turbo PASCAL?

❷ Cum se defineşte un tip procedural? Care este mulţimea de valori ale unui tip procedural?

❸ Corectaţi programul:

```
Program P103;
 { Eroare }
 type TF=function(x : real) : real;

 function F1(x : real) : real;
 begin
   F1:=sqr(x);
 end; { F1 }

 procedure T(F : TF; x : real);
 begin
   writeln('x=', x, '   f(x)=', F(X));
 end; { T }

 begin
  T(F1, 2);
  T(sqrt, 2);
  readln;
 end.
```

❹ Utilizînd procedura `Tabel` din programul `P102`, elaboraţi un program care tabelează funcţiile:

a) $x^3 - 4, 2x + 1, 6x$;

b) $x^2 + x - 1, x + 1, \sin x$;

c) $\sin x, \cos x, \mathrm{tg}\, x$;

d) x, x^2, x^3.

❺ Precizaţi ce va afişa pe ecran programul ce urmează:

```
Program P104;
  { Parametru formal procedură }
```

```
type TP=procedure(i : integer);
var v : TP;

procedure P1(j : integer);
begin
  writeln(j+1);
end; { P1 }

procedure P2(k : integer);
begin
  writeln(k+2);
end; {  P2 }

procedure P3(m : integer);
begin
  writeln(m+3);
end; {  P3 }·

procedure T(P : TP; n : integer);
begin
  P(n);
end; { T }

begin
 v:=P3;
 T(v, 1);
 v:=P2;
 T(v, 1);
 T(P1, 1);
 readln;
end.
```

❻ Elaboraţi o procedură care afişează pe ecran tabelele funcţiilor de două variabile reale cu rezultate reale. Tabelaţi funcţiile:

a) $z = x + y,\ \ z = x - y,\ \ z = y - x$;

b) $z = \cos(x + y),\ \ z = \sin(x + y),\ \ z = tg(x + y)$;

c) $z = \begin{cases} x + y, & x \geq y \ ; \\ x - y, & x < y \ ; \end{cases}$

$z = \begin{cases} x^2 + y^{2,} & x + y \geq 2 \ ; \\ x^2 - y^{2,} & x + y < 2 \ ; \end{cases}$

$z = \begin{cases} \sin(x + y), & x \geq y \ ; \\ \cos(x - y), & x < y \ ; \end{cases}$

❼ Funcţia $y = f(x)$ este nenulă în punctele 1, 2, ..., n. Elaboraţi o procedură care determină numărul schimbărilor de semn în şirul $f(1), f(2), ..., f(n)$.

❽ Se consideră mulţimea tuturor funcţiilor $f(i)$ de o variabilă întreagă cu rezultate reale. Elaboraţi o funcţie care calculează $f(1) + f(2) + ... +f(n)$.

❾ Se consideră funcţia $f(i)$ de o variabilă întreagă cu rezultate întregi. Elaboraţi o funcţie care calculează de cîte ori şirul $f(1), f(2), ..., f(n)$ conţine subşirul 1, 0, 1.

⑩ Ecuaţia $f(x) = 0$ are în intervalul (a, b) o singură rădăcină. Elaboraţi o procedură pentru determinarea rădăcinii ecuaţiei în studiu prin metoda înjumătăţirii.
Indicaţie: Se calculează $c = (a + b)/2$. Analizînd semnele valorilor $f(a)$, $f(c)$ şi $f(b)$ se selectează unul din intervalele (a, c), (c, b) ce conţine rădăcina ecuaţiei. Intervalul selectat se împarte din nou în două intervale ş.a.m.d. pînă se ajunge la precizia dorită.

5.5. DOMENII DE VIZIBILITATE

Corpul unui program sau subprogram se numeşte **bloc**. Deoarece subprogramele sînt incluse în programul principal şi pot conţine la rîndul lor alte subprograme, rezultă că blocurile pot fi **imbricate** (in-

```
Program P105;
{ Structura de bloc a programului }    { nivel 0}
var a: real;
{ 1}
procedure P(b : real);
var c : real;                          { nivel 1}
{ 2}
procedure Q(d : integer);
{ 3}                                   { nivel 2}
var c : char;
{ 4}
begin
c:=chr(d);
writeln('În procedura Q c = ', c);
end; { 5}
begin
writeln('b=', b);
c:=b+1;
writeln('În procedura P c = ', c);
Q(35);
end; { 6}

function F(x : real) : real;
begin                                  { nivel 1}
f:=x/2;
end;

begin
a:=F(5);
writeln('a=', a);
P(a);
readln;
end { 7} .
```

Fig. 5.2. Structura de bloc a unui program PASCAL

cluse unul în altul). Această imbricare de blocuri este denumită **structura de bloc** a programului PASCAL.

Într-o structură fiecărui bloc i se ataşează cîte un nivel de imbricare. Programul principal este considerat de nivel 0, un bloc definit în programul principal este de nivel 1. În general, un bloc definit în nivelul *n* este de nivelul *n* + 1.

Pentru exemplificare, în *fig. 5.2* este prezentată structura de bloc a programului `P105`.

De regulă, un bloc PASCAL include declaraţii de etichete, variabile, funcţii, parametri ş.a.m.d. O declaraţie introduce un nume, care poate fi o etichetă sau un identificator. O declaraţie dintr-un bloc poate redefini un nume declarat în exteriorul lui. În consecinţă, în diferite părţi ale programului unul şi acelaşi nume poate desemna obiecte diferite.

Prin **domeniul de vizibilitate** al unei declaraţii se înţelege textul de program, în care numele introdus desemnează obiectul specificat de declaraţia în studiu. Domeniul de vizibilitate începe imediat după terminarea declaraţiei şi se sfîrşeşte o dată cu textul blocului respectiv. Deoarece blocurile pot fi imbricate, domeniul de vizibilitate nu este neapărat o porţiune continuă din textul programului. Domeniul de vizibilitate al unei declaraţii dintr-un bloc inclus **acoperă** domeniul de vizibilitate al declaraţiei ce implică acelaşi nume din blocul exterior.

De exemplu, în programul `P105` domeniul de vizibilitate al declaraţiei **var** `a : integer` este textul cuprins între punctele marcate { 1} şi { 7} . Domeniul de vizibilitate al declaraţiei **var** `c : real` este format din două fragmente de text cuprinse între { 2} , { 3} şi { 5} , { 6} . Domeniul de vizibilitate al declaraţiei **var** `c : char` este textul cuprins între { 4} şi { 5} .

Cunoaşterea domeniilor de vizibilitate ale declaraţiilor este necesară pentru determinarea obiectului curent desemnat de un nume.

De exemplu, identificatorul `c` din instrucţiunea

```
c:=chr(d)
```

a programului `P105` desemnează o variabilă de tip `char`. Acelaşi identificator din instrucţiunea

```
c:=b+1
```

desemnează o variabilă de tip `real`.

De reţinut că declaraţia unui nume de funcţie/procedură se consideră terminată la sfîrşitul antetului. Prin urmare, domeniul de vizibilitate al unei astfel de declaraţii include şi corpul funcţiei/procedurii respective. Acest fapt face posibil **apelul recursiv**: în corpul funcţiei/procedurii aceasta poate fi referită, fiind vizibilă. Evident, decla-

rația unui parametru formal este vizibilă numai în corpul subprogramului respectiv.

De exemplu, domeniul de vizibilitate al declarației **procedure** Q este textul cuprins între punctele marcate { 3} și { 6} . Domeniul de vizibilitate al declarației `d:integer` este textul cuprins între { 3} și { 5} .

Întrebări și exerciții

❶ Cum se determină domeniul de vizibilitate al unei declarații?

❷ Determinați domeniile de vizibilitate ale declarațiilor `b : real` și `x : real` din programul P105 (*fig 5.2*).

❸ Precizați structura de bloc a programului ce urmează. Indicați domeniul de vizibilitate al fiecărei declarații și determinați obiectele desemnate de fiecare apariție a identificatorilor c și x.

```
Program P106;
  { Redefinirea constantelor }
 const c=1;

  function F1(x : integer) : integer;
 begin
  F1:=x+c;
 end; { F1 }

  function F2(c : real) : real;
 const x=2.0;
 begin
  F2:=x+c;
 end; { F2 }

  function F3(x : char) : char;
 const c=3;
 begin
  F3:=chr(ord(x)+c);
 end; { F3 }
  begin
  writeln('F1=', F1(1));
  writeln('F2=', F2(1));
  writeln('F3=', F3('1'));
  readln;
 end.
```

Ce va afișa pe ecran programul în studiu?

❹ Determinați domeniile de vizibilitate ale identificatorilor P și F din programul P105 (*fig. 5 2*).

❺ Comentați programul ce urmează:

```
Program P107;
  { Eroare }
  var a : real;
```

```
 procedure P(x : real);
var a : integer;
begin
 a:=3.14;
 writeln(x+a);
end; { P }
 begin
 a:=3.14;
 P(a);
end.
```

❻ Cum se determină obiectul desemnat de apariţia unui nume într-un program PASCAL?

5.6. COMUNICAREA PRIN VARIABILE GLOBALE

Execuţia unui apel de subprogram presupune transmiterea datelor de prelucrat funcţiei sau procedurii respective. După executarea ultimei instrucţiuni din subprogram, rezultatele produse trebuie întoarse în locul de apel. Cunoaştem deja că datele de prelucrat şi rezultatele produse pot fi transmise prin parametri. Parametrii formali se specifică în antetul funcţiei sau procedurii, iar parametrii actuali — în locul apelului.

În completare la modul de transmitere a datelor prin parametri, limbajul PASCAL permite comunicarea prin variabile globale.

O variabilă este globală **relativ la un subprogram** atunci cînd ea este declarată în programul sau subprogramul ce îl cuprinde fără să fie redeclarată în subprogramul în studiu. Întrucît variabilele globale sînt cunoscute atît în subprogram, cît şi în afara lui, ele pot fi folosite pentru transmiterea datelor de prelucrat şi returnarea rezultatelor.

Exemplu:

```
Program P108;
  { Comunicarea prin variabile globale }
 var a,              { variabilă globală în P }
     b : real;       { variabilă globală în P, F }

 procedure P;
 var c : integer;    { variabilă locală în P }
 begin
  c:=2;
  b:=a*c;
 end; { P }

 function F : real;
 var a : 1..5;       { variabilă locală în F }
```

```
begin
 a:=3;
 F:=a+b;
end; { F }

begin
 a:=1;
 P;
 writeln(b);      { se afişează 2.0000000000E+00 }
 writeln(F);      { se afişează 5.0000000000E+00 }
 readln;
end.
```

Datele de prelucrat se transmit procedurii P prin variabila globală a. Rezultatul produs de procedură se returnează în blocul de apel prin variabila globală b. Valoarea argumentului funcţiei F se transmite prin variabila globală b. Menţionăm că variabila a este locală în F şi nu poate fi folosită pentru transmiterea datelor în această funcţie.

De obicei, comunicarea prin variabile globale se utilizează în cazurile în care mai multe subprograme prelucrează aceleaşi date. Pentru exemplificare amintim funcţiile cu argumente de tip *tablou*, procedurile care prelucrează tablouri şi fişiere de angajaţi, persoane, elevi etc.

Întrebări şi exerciţii

❶ Explicaţi termenii *variabilă globală relativ la un subprogram* şi *variabilă locală într-un subprogram.*

❷ Numiţi variabilele globale şi variabilele locale din programul P105 (*fig. 5.2*).

❸ Poate fi oare o variabilă locală în acelaşi timp şi o variabilă globală relativ la un subprogram?

❹ Numiţi variabilele globale şi variabilele locale din programul ce urmează. Ce va afişa pe ecran acest program?

```
Program P109;
 { Comunicarea prin variabile globale }
 var a : integer;

 procedure P;
 var b, c, d : integer;

 procedure Q;
  begin
  c:=b+1;
  end; { Q }

 procedure R;
  begin
  d:=c+1;
  end; { R }
```

```
begin
 b:=a;
 Q;
 R;
 a:=d;
end; { P }

begin
 a:=1;
 P;
 writeln(a);
 readln;
end.
```

❺ Se consideră declaraţiile

```
Type  Ora=0..23;
      Grade=-40..+40;
      Temperatura=array[Ora] of Grade;
```

Componentele unei variabile de tip `Temperatura` reprezintă temperaturile măsurate din oră în oră pe parcursul a 24 de ore. Elaboraţi o pocedură care:

a) indică maximumul şi minimumul temperaturii;

b) indică ora (orele) la care s-a înregistrat o temperatură maximă;

c) înscrie ora (orele) la care s-a înregistrat o temperatură minimă într-un fişier *text*.

Comunicarea cu procedurile respective se va face prin variabile globale.

❻ Se consideră fişiere arbitrare de tip *text*. Elaboraţi o funcţie care:

a) returnează numărul de linii dintr-un fişier;

b) calculează numărul de vocale dintr-un text;

c) calculează numărul de cuvinte dintr-un text (cuvintele reprezintă şiruri de caractere separate prin spaţiu sau sfîrşit de linie);

d) returnează lungimea medie a liniilor din text;

e) calculează lungimea medie a cuvintelor din text;

f) returnează numărul semnelor de punctuaţie din text.

Comunicarea cu funcţiile respective se va face prin variabile globale.

5.7. EFECTE COLATERALE

Destinaţia unei funcţii este să întoarcă ca rezultat o singură valoare. În mod obişnuit, argumentele se transmit funcţiei prin parametri-valoare, iar rezultatul calculat se returnează în locul de apel prin numele funcţiei. În completare, limbajul PASCAL permite transmiterea argumentelor prin variabile globale şi parametri-variabilă.

Prin **efect colateral** se înţelege o atribuire (în corpul funcţiei) a unei valori la o variabilă globală sau la un parametru formal variabilă.

Efectele colaterale pot influenţa în mod neaşteptat execuţia unui program şi complică procesele de depanare.

Prezentăm în continuare exemple defectuoase de programare care folosesc funcţii cu efecte colaterale.

```
Program P110;
 {Efect colateral - atribuire la o variabilă globală}
 var a : integer; { variabilă globală }

 function F(x : integer) : integer;
  begin
    F:=a*x;
    a:=a+1;      {atribuire defectuoasă }
 end; { F }
  begin
    a:=1;
    writeln(F(1)); { se afişează 1 }
    writeln(F(1)); { se afişează 2 }
    writeln(F(1)); { se afişează 3 }
    readln;
  end.
```

În programul P110 funcţia F returnează valoarea expresiei a*x. Pe lîngă asta însă, atribuirea a:=a+1 alterează valoarea variabilei globale a. În consecinţă, pentru una şi aceeaşi valoare 1 a argumentului x funcţia returnează rezultate diferite, fapt ce nu se încadrează în conceptul uzual de funcţie.

```
Program P111;
 {Efect colateral - atribuire la un parametru formal}
 var a : integer;

  function F(var x : integer) : integer;
    begin
      F:=2*x;
      x:=x+1;     { atribuire defectuoasă }
    end; { F }

 begin
  a:=2;
  writeln(F(a));      { se afişează 4 }
  writeln(F(a));      { se afişează 6 }
  writeln(F(a));      { se afişează 8 }
  readln;
 end.
```

În programul P111 funcţia F returnează valoarea expresiei 2*x. Întrucît x este un parametru formal variabilă, atribuirea x:=x+1 schimbă valoarea parametrului actual din apel, şi anume, a variabilei

a din programul principal. Faptul că apelurile textual identice `F(a)`, `F(a)` şi `F(a)` returnează rezultate ce diferă, poate crea confuzii în procesul depanării.

În cazul procedurilor, atribuirile asupra variabilelor globale produc efecte colaterale similare celor discutate pentru astfel de atribuiri la funcţii. Întrucît mijlocul-standard de întoarcere de rezultate din procedură este prin parametri formali variabilă, atribuirile asupra unor astfel de parametri nu sînt considerate ca efecte colaterale.

Efectele colaterale introduc abateri de la procesul-standard de comunicare, prin care variabilele participante sînt desemnate explicit ca parametri formali în declaraţie şi parametri actuali în apel. Consecinţele efectelor colaterale se pot propaga în domeniul de vizibilitate al declaraţiilor globale şi pot interfera cu cele similare produse la execuţia altor proceduri şi funcţii. În astfel de condiţii, utilizarea variabilelor globale devine riscantă. Prin urmare, la elaborarea programelor complexe se vor aplica următoarele recomandări:

✓ Comunicarea funcţiilor cu mediul de chemare se va face prin transmiterea de date spre funcţie prin parametri formali valoare şi întoarcerea unui singur rezultat prin numele ei.

✓ Comunicarea procedurilor cu mediul de chemare se va face prin transmiterea de date prin parametri formali valoare sau variabilă şi întoarcerea rezultatelor prin parametri formali variabilă.

✓ Variabilele globale pot fi folosite pentru transmiterea datelor în subprograme, însă valorile lor nu trebuie să fie schimbate de acestea.

Întrebări şi exerciţii

❶ Care este cauza efectelor colaterale? Ce consecinţe pot avea aceste efecte?

❷ Precizaţi ce vor afişa pe ecran programele ce urmează:

```
Program P112;
  { Efecte colaterale }
 var a, b : integer;

  function F(x : integer) : integer;
    begin
    F:=a*x;
    b:=b+1;
    end; { F }

  function G(x : integer) : integer;
    begin
     G:=b+x;
     a:=a+1;
    end; { G }
```

```
begin
  a:=1; b:=1;
  writeln(F(1));
  writeln(G(1));
  writeln(F(1));
  writeln(G(1));
  readln;
end.
```

```
Program P113;
 { Efecte colaterale }
 var a : integer;
     b : real;

  function F(var x : integer) : integer;
    begin
      F:=x;
      x:=x+1;
    end; { F }

  procedure P(x,y:integer; var z:real);
    begin
      z:=x/y;
    end; { P }

begin
  a:=1;
  P(F(a), a, b);
  writeln(a, '   ', b);
  readln;
  end.
```

```
Program P114;
  { Efecte colaterale }
 var a, b : real;

  procedure P(var x, y : real);
  { Interschimbarea valorilor variabilelor x, y }
    begin
      a:=x;
      x:=y;
      y:=a;
    end; { P }

begin
  a:=1; b:=2;
  P(a, b);
  writeln(a, b);
  a:=3; b:=4;
  P(a, b);
```

```
  writeln(a, b);
  readln;
end.
```

❸ Cum pot fi evitate efectele colaterale?

5.8. RECURSIA

Recursia se defineşte ca o situaţie în care un subprogram se autoapelează fie direct, fie prin intermediul altei funcţii sau proceduri. Subprogramul care se autoapelează se numeşte ***recursiv***.

De exemplu, presupunem că este definit tipul

```
type Natural = 0..MaxInt;
```

Funcţia *factorial*

$$f(n)=\begin{cases}1, & \text{dacă } n=0;\\ n\cdot f(n-1), & \text{dacă } n>0\end{cases}$$

poate fi exprimată în PASCAL, urmînd direct definiţia, în forma

```
function F(n : Natural) : Natural;
begin
    if n=0 then F:=1
    else F:=n*F(n-1)
end; { F}
```

Efectul unui apel `F(7)` este declanşarea unui lanţ de apeluri ale funcţiei `F` pentru parametrii actuali `6, 5, ..., 2, 1, 0`:

```
F(7) -> F(6) -> F(5) -> ... -> F(1) -> F(0).
```

Apelul `F(0)` determină evaluarea directă a funcţiei şi oprirea procesului repetitiv; urmează revenirile din apeluri şi evaluarea lui `F` pentru `1, 2, ..., 6, 7`, ultima valoare fiind întoarcerea în locul primului apel.

Funcţia

$$\mathrm{fib}(n)=\begin{cases}0, & \text{dacă } n=0;\\ 1, & \text{dacă } n=1;\\ \mathrm{fib}(n-1)+\mathrm{fib}(n-2), & \text{dacă } n>1\end{cases}$$

are ca valori numerele lui Fibonacci. Urmînd definiţia, obţinem:

```
function Fib(n:Natural):Natural;
begin
  if n=0 then Fib:=0
  else if n=1 then Fib:=1
```

```
        else Fib:=Fib(n-1)+Fib(n-2)
end; { Fib}
```

Fiecare apel al funcţiei `Fib` pentru `n > 1` generează două apeluri `Fib(n-1)`, `Fib(n-2)` ş.a.m.d., de exemplu:

```
Fib(4) -> Fib(3), Fib(2) -> Fib(2),
Fib(1), Fib(1), Fib(0) -> Fib(1), Fib(0).
```

Din exemplele în studiu se observă că recursia este utilă pentru programarea unor calcule repetitive. Repetiţia este asigurată prin execuţia unui subprogram care conţine un apel la el însuşi: cînd execuţia ajunge la acest apel, este declanşată o nouă execuţie ş.a.m.d.

Evident, orice subprogram recursiv trebuie să includă condiţii de oprire a procesului repetitiv. De exemplu, în cazul funcţiei *factorial* procesul repetitiv se opreşte cînd *n* ia valoarea 0; în cazul funcţiei `Fib` procesul se opreşte cînd *n* ia valoarea 0 sau 1.

La orice apel de subprogram în memoria calculatorului vor fi depuse următoarele informaţii:

— valorile curente ale parametrilor transmişi prin valoare;
— locaţiile (adresele) parametrilor-variabilă;
— adresa de retur, adică adresa instrucţiunii ce urmează după apel.

Prin urmare, la apeluri recursive spaţiul ocupat din memorie va creşte rapid, riscînd depăşirea capacităţii de memorare a calculatorului. Astfel de cazuri pot fi evitate, înlocuind recursia prin iteraţie (instrucţiunile **for**, **while**, **repeat**). Pentru exemplificare prezentăm o formă nerecursivă a funcţiei *factorial*:

```
function F(n: Natural): Natural;
var i, p : Natural;
begin
    p:=1;
      for i:=1 to n do p:=p*i;
    F:=p;
end; { F}
```

Recursia este deosebit de utilă în cazurile în care elaborarea unor algoritmi nerecursivi este foarte complicată: translatarea programelor PASCAL în limbajul cod-maşină, grafica pe calculator, recunoaşterea formelor ş.a.

Întrebări şi exerciţii

❶ Cum se execută un subprogram recursiv? Ce informaţii se depun în memoria calculatorului la execuţia unui apel recursiv?
❷ Care este diferenţa dintre recursie şi iteraţie?
❸ Elaboraţi o formă nerecursivă a funcţiei lui Fibonacci.
❹ Scrieţi un subprogram recursiv care:

a) calculează suma $S(n) = 1 + 3 + 5 + ... + (2n - 1)$;
b) calculează produsul $P(n) = 1 \times 4 \times 7 \times ... \times (3n - 2)$;
c) inversează un şir de caractere;
d) calculează produsul $P(n) = 2 \times 4 \times 6 \times ... \times 2n$.

❺ Elaboraţi un program care citeşte de la tastatură numerele naturale *m, n* şi afişează pe ecran valoarea funcţiei lui Ackermann:

$$a(m,n) = \begin{cases} n+1, & \text{dacă } m=0 \text{ ;} \\ a(m-1,1), & \text{dacă } n=0 \text{ ;} \\ a(m-1,a(m,n-1)), & \text{dacă } m>0 \text{ şi } n>0. \end{cases}$$

Calculaţi a(0, 0), a(1, 2), a(2, 1) şi a(2, 2). Încercaţi să calculaţi a(4, 4) şi a(10, 10). Explicaţi mesajele afişate pe ecran.

❻ Se consideră declaraţia

```
type Vector=array [ 1..20] of integer;
```

Elaboraţi un subprogram recursiv care:

a) afişează componentele vectorului pe ecran;
b) calculează suma componentelor;
c) inversează componentele vectorului;
d) calculează suma componentelor pozitive;
e) verifică dacă cel puţin o componentă a vectorului este negativă;
f) calculează produsul componentelor negative;
g) verifică dacă cel puţin o componentă a vectorului este egală cu un număr dat.

❼ Elaboraţi o formă nerecursivă a funcţiei ce urmează:

```
function S(n:Natural):Natural;
 begin
  if n=0 then S:=0
  else S:=n+S(n-1)
 end; { S}
```

❽ Scrieţi o funcţie recursivă care returnează valoarea `true` dacă şirul de caractere `s` este conform definiţiei

<Număr> ::= *<Cifră>* | *<Cifră>* *<Număr>*

Indicaţie. Forma unei astfel de funcţii derivă din formula metalingvistică. Varianta nerecursivă

```
function N(s : string) : boolean;
var i : integer;
    p : boolean
begin
    p:=(s<>'');
    for i=1 to length(s) do
    p:=p and (s[ i] in [ '0'..'9'] );
    N:=p;
end;
```

derivă din definiţia

<Număr> ::= *<Cifră>* {*<Cifră>*}

❾ Se consideră următoarele formule metalingvistice:

<Număr> ::= *<Cifră>* {*<Cifră>*}
<Semn> ::= + | –
<Expresie> ::= *<Număr>* | *<Expresie>* *<Semn>* *<Expresie>*

Scrieţi o funcţie recursivă care returnează valoarea `true` dacă şirul de caractere `s` este conform definiţiei unităţii lexicale *<Expresie>*.

5.9. DECLARAŢII ANTICIPATE

Toate subprogramele recursive prezentate pînă acum includ în partea executabilă un autoapel, motiv pentru care se spune că s-a folosit ***recursia directă***. Există însă situaţii în care un subprogram trebuie autoapelat prin intermediul altui subprogram, adică apare ***recursia indirectă***.

Exemplu:

$$f(x)=\begin{cases} 1, & \text{dacă } x=0; \\ x+g(x-1), & \text{dacă } x>0; \end{cases}$$

$$g(x)=\begin{cases} 2, & \text{dacă } x=0; \\ x+f(x-1), & \text{dacă } x>0. \end{cases}$$

Evident, în textul PASCAL al funcţiei $f(x)$ se va utiliza apelul $g(x-1)$, iar în textul $g(x)$ se va utiliza apelul $f(x-1)$.

Conform regulilor de bază ale limbajului PASCAL, locul de definiţie al unui nume trebuie să precede textual utilizările numelui respectiv. În cazul funcţiilor $f(x)$ şi $g(x)$, oricare ar fi forma textuală a programului, numele uneia dintre funcţii apare înaintea locului de definire. Pentru a soluţiona astfel de probleme, se utilizează declaraţiile anticipate de subprograme.

O **declaraţie anticipată** include antetul subprogramului şi directiva **`forward`** (înainte). Corpul subprogramului apare textual undeva mai jos şi este precedat de un antet redus în care se specifică doar numele subprogramului, fără parametri şi tipuri.

Exemplu:

```
Program P115;
  { Declaraţii anticipate }
 type Natural=0..MaxInt;

 function F(x : Natural) : Natural;
```

```
  forward;  { declaraţia anticipată a funcţiei F }

  function G(x : Natural) : Natural;
    begin
      if x=0 then G:=2
         else G:=x+F(x-1);
    end; { G }

  function F;
    begin       { corpul funcţiei F }
      if x=0 then F:=1
         else F:=x+G(x-1);
    end; { F }

begin
  writeln(F(5));
  readln;
end.
```

În principiu, declaraţiile anticipate pot fi folosite pentru definirea oricăror proceduri sau funcţii, nu neapărat recursive. Menţionăm însă că utilizarea nejustificată a declaraţiilor anticipate complică structura de bloc a programelor PASCAL.

Întrebări şi exerciţii

❶ Care este diferenţa dintre *recursia directă* şi *recursia indirectă*?

❷ Este oare necesară utilizarea directivei **forward** în cazul recursiei indirecte? Argumentaţi răspunsul.

❸ Elaboraţi un program care citeşte de la tastatură numărul natural n şi afişează pe ecran valoarea funcţiei $f(n)$:

$$f(n)=\begin{cases} 1, & \text{dacă } n=0 ;\\ g(n-1)+h(n-1), & \text{dacă } n>0 ;\end{cases}$$

$$g(n)=\begin{cases} 2, & \text{dacă } n=0 ;\\ f(n-1)+h(n-1), & \text{dacă } n>0 ;\end{cases}$$

$$h(n)=\begin{cases} 3, & \text{dacă } n=0;\\ f(n-1)+g(n-1), & \text{dacă } n>0.\end{cases}$$

Calculaţi valorile $f(0)$, $f(5)$, $f(10)$ şi $f(15)$. Încercaţi să calculaţi $f(100)$ şi $f(200)$. Explicaţi mesajele afişate pe ecran.

❹ Precizaţi ce va afişa pe ecran programul ce urmează:

```
Program P116;
 var x : real;

function F(x : real) : real; forward;
```

```
procedure P(var x : real); forward;
procedure Q(var x : real); forward;

  function F;
    begin
      F:=2*x
    end;

  procedure P;
  begin
    x:=x+1
  end;

  procedure Q;
    begin
      x:=x+2
    end;

begin
  x:=1;
  P(x);
  Q(x);
  writeln(F(x));
  readln;
end.
```

5.10. SINTAXA DECLARAŢIILOR ŞI APELURILOR DE SUBPROGRAME

În general, definirea unei funcţii se face cu ajutorul următoarelor formule metalingvistice:

<Funcţie> ::= *<Antet funcţie> ;* <Corp> | *<Antet funcţie> ; <Directivă>* | **function** *<Identificator>*; <Corp>

<Antet funcţie> ::= **function** *<Identificator>* [*<Listă parametri formali>*] : *<Identificator>*

Diagramele sintactice sînt prezentate în *fig. 5.3*.

Procedurile se definesc cu ajutorul următoarelor formule:

<Procedură>::=*<Antet procedură> ;*<Corp> | *<Antet procedură> ; <Directivă>* | **procedure** *<Identificator> ;*<Corp>

<Antet procedură> := **procedure** *<Identificator>* [*<Listă parametri formali>*]

Diagramele sintactice sînt prezentate în *fig. 5.4*.

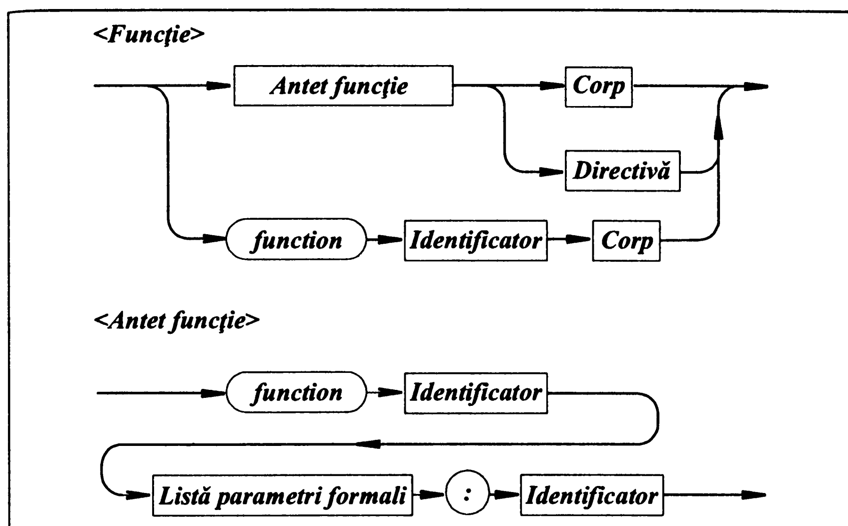


Fig. 5.3. Sintaxa declaraţiilor de funcţii

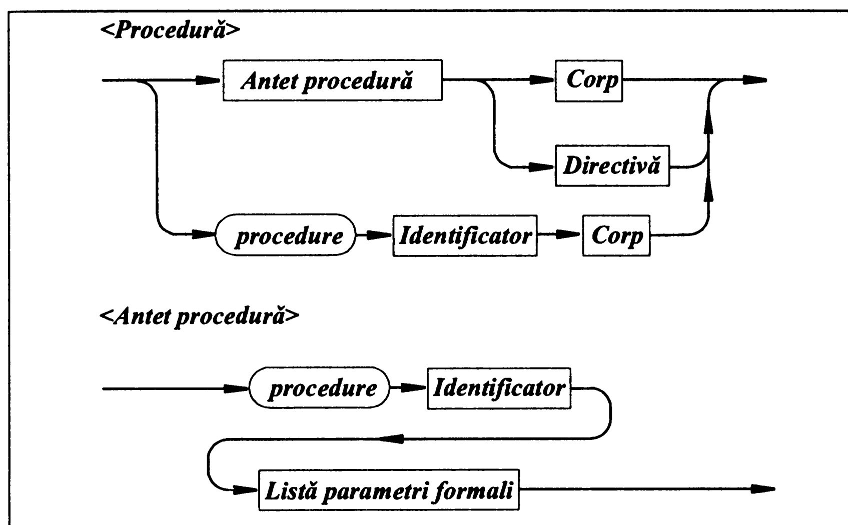


Fig. 5.4. Sintaxa declaraţiilor de proceduri

Listele de parametri formali au următoarea sintaxă:

<Listă parametri formali> ::=
(*<Parametru formal>* {; *<Parametru formal>*})

<*Parametru formal*> ::= [**var**] <*Identificator*> {, <*Identificator*>}
: <*Identificator*> | <*Antet funcţie*> |
<*Antet procedură*>

Diagrama sintactică este prezentată în *fig. 5.5*.

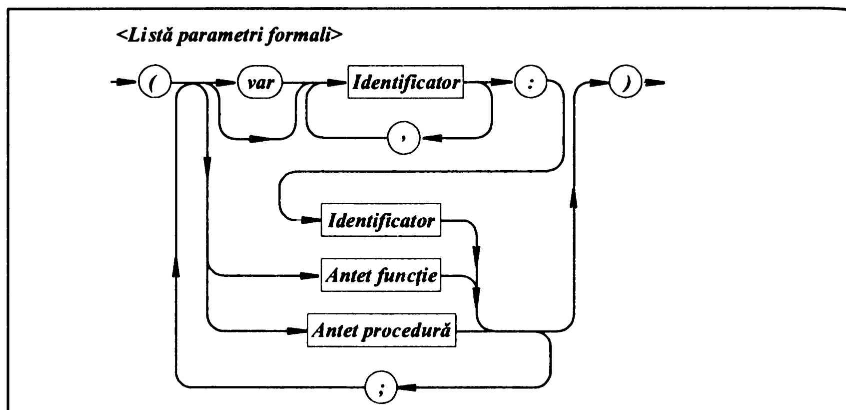


Fig. 5.5. Diagrama sintactică <*Lista parametri formali*>

Amintim că în lipsa cuvîntului-cheie **var** identificatorii din listă specifică **parametrii-valoare**. Cuvîntul **var** prefixează **parametrii-variabilă**. Antetul unei funcţii (proceduri) din listă specifică un **parametru-funcţie (procedură)**. În TurboPASCAL astfel de parametri se declară explicit ca aparţinînd unui **tip procedural** şi au forma parametrilor-valoare. Limbajul PASCAL extinde sensul uzual al noţiunii de funcţie, permiţînd returnarea valorilor nu numai prin numele funcţiei, ci şi prin parametri-variabilă.

Un **apel de funcţie** are forma:

<*Apel funcţie*> ::= <Nume funcţie> [<*Listă parametri actuali*>]

iar o instrucţiune **apel de procedură**:

<*Apel procedură*> ::= <Nume procedură> [<*Listă parametri actuali*>]

Parametrii actuali se specifică cu ajutorul formulelor:

<*Listă parametri actuali*> ::=
(<*Parametru actual*> {, <*Parametru actual*>})

<*Parametru actual*> ::= <Expresie> | <*Variabilă*> |
<*Nume funcţie*> | <*Nume procedură*>

Diagrama sintactică este prezentată în *fig. 5.6*.

Corespondenţa între un parametru actual şi parametrul formal se face prin poziţia ocupată de aceştia în cele două liste.

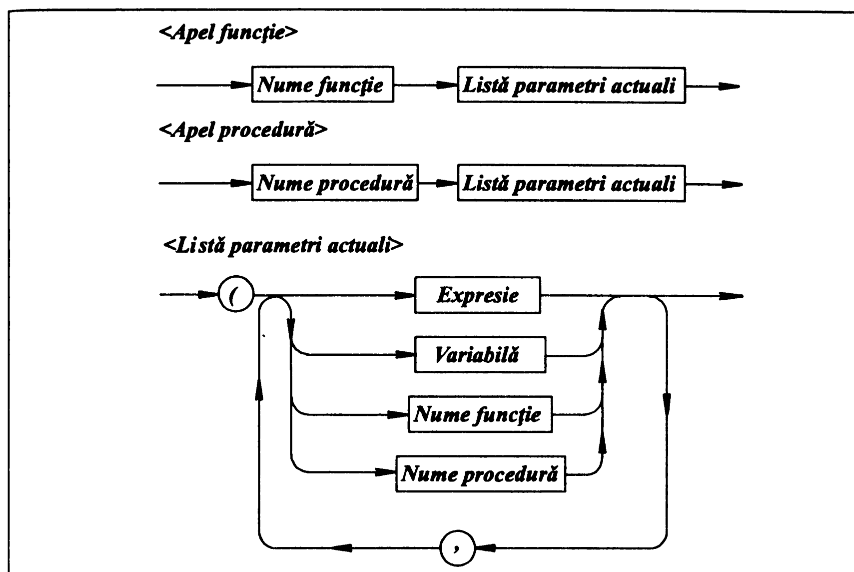


Fig. 5.6. Sintaxa apelurilor de funcţii şi proceduri

În cazul unui **parametru-valoare** drept parametru actual poate fi utilizată orice expresie, în particular, o constantă sau o variabilă. Expresia respectivă trebuie să fie compatibilă din punct de vedere al atribuirii cu tipul parametrului formal. Modificările parametrilor-valoare nu se transmit în exteriorul subprogramului.

În cazul unui **parametru-variabilă** drept parametri actuali pot fi utilizate numai variabile. Modificările parametrilor-variabilă se transmit în exteriorul subprogramului.

În cazul unui **parametru-funcţie (procedură)** drept parametru actual poate fi utilizat orice nume de funcţie (procedură) antetul căreia are forma specificată în lista parametrilor formali.

Întrebări şi exerciţii

❶ Cînd se utilizează declaraţiile de forma

`function` *<Identificator>* ;*<Corp>* ?

❷ Indicaţi pe diagramele sintactice din *fig. 5.3* şi *5.5* drumurile care corespund declaraţiilor de funcţii din programul `P115`, paragraful 5.9.

❸ Care este diferenţa dintre un parametru-valoare şi un parametru-variabilă?

❹ Indicaţi pe diagramele sintactice din *fig. 5.4* şi *5.5* drumurile care corespund declaraţiilor de proceduri din programul `P116`, paragraful 5.9.

❺ Cum se specifică parametrii formali funcţie (procedură) în PASCAL? În Turbo PASCAL?

❻ Indicaţi pe diagramele sintactice din *fig. 5.3–5.6* drumurile care corespund declaraţiilor şi apelurilor de subprograme din programul `P102`, paragraful 5.4.

Capitolul 6

STRUCTURI DINAMICE DE DATE

6.1. VARIABILE DINAMICE. TIPUL *REFERINȚĂ*

Variabilele declarate în secțiunea **var** a unui program sau subprogram se numesc **variabile statice**. Numărul variabilelor statice se stabilește în momentul scrierii programului și nu poate fi schimbat în timpul execuției. Există însă situații în care numărul necesar de variabile nu este cunoscut din timp.

De exemplu, presupunem că este necesară prelucrarea datelor referitoare la persoanele care formează un șir de așteptare (o coadă) la o casă de bilete. Lungimea cozii este nedefinită. De fiecare dată cum apare o persoană nouă, trebuie să se creeze o variabilă de tipul respectiv. După ce persoana pleacă, variabila corespunzătoare devine inutilă.

Variabilele care sînt create și eventual distruse în timpul execuției programului se numesc **variabile dinamice**.

Accesul la variabilele dinamice se face prin intermediul variabilelor de tip *referință*. De obicei, un tip *referință* se definește printr-o declarație de forma:

type $T_r = \hat{}T_b;$

unde T_r este numele tipului *referință*, iar T_b este tipul de bază. Semnul "^" se citește "adresă". Evident, pot fi utilizate și tipuri referință anonime. Diagrama sintactică <*Tip referință*> este prezentată în *fig. 6.1.*

Mulțimea de valori ale unui tip de date *referință* constă din adrese.

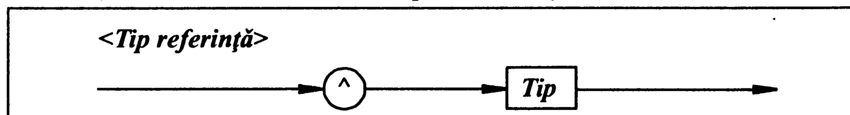


Fig. 6.1. Diagrama sintactică <*Tip referință*>

Fiecare adresă identifică o variabilă dinamică ce aparține tipului de bază. La această mulțime de adrese se mai adaugă o valoare specială, notată **nil** (zero) care nu identifică nici o variabilă.

Exemplu:

```
type AdresaInteger=^integer;
     AdresaChar=^char;
var  i : AdresaInteger;
     r : ^real;
     c : AdresaChar;
```

Valoarea curentă a variabilei i va indica o variabilă dinamică de tipul integer. Într-un mod similar, variabilele de tip referință r şi c identifică variabile de tipul real şi, respectiv, char. Subliniem faptul că tipurile de date AdresaInteger, AdresaChar şi tipul anonim ^real sînt tipuri *referință* distincte.

Operațiile care se pot face cu valorile unui tip de date *referință* sînt = şi <>. Valorile de tip *referință* nu pot fi citite de la tastatură şi afişate pe ecran.

Crearea unei variabile dinamice se realizează cu procedura predefinită new (nou). Apelul acestei proceduri are forma

```
new(p)
```

unde p este o variabilă de tip referință.

Procedura alocă spațiu de memorie pentru variabila nou creată şi returnează adresa zonei respective prin variabila p. În continuare variabila dinamică poate fi accesată prin aşa-zisa **dereperare**: numele variabilei de tip *referință* p este urmat de semnul de "^". Dereperarea unei variabile de tip *referință* cu conținutul **nil** va declanşa o eroare de execuție.

Exemplu:

new (i); i^:=1 — crearea unei variabile dinamice de tipul integer; variabilei create i se atribuie valoarea 1;

new(r); r^:=2.0 — crearea unei variabile dinamice de tipul real; variabilei create i se atribuie valoarea 2.0;

new(c); c^:='*' — crearea unei variabile dinamice de tipul char; variabilei create i se atribuie valoarea '*'.

Subliniem faptul că variabila dinamică p^ obținută printr-un apel new(p) este distinctă de toate variabilele create anterior. Prin urmare, executarea instrucțiunilor

```
new(p); new(p); ...; new(p)
```

conduce la crearea unui şir v_1, v_2, ..., v_n de variabile dinamice. Numai ultima variabilă creată, v_n, este referită prin p^. Întrucît valorile variabilelor de tip *referință* reprezintă adresele anumitor zone din memoria internă a calculatorului, variabilele în studiu se numesc **indicatori de adresă**.

Distrugerea unei variabile dinamice şi eliberarea zonei respective de memorie se realizează cu procedura predefinită `dispose` (a dispune). Apelul acestei proceduri are forma:

```
dispose(p)
```

unde `p` este o variabilă de tip *referinţă*.

Exemple:

```
dispose(i); dispose(r); dispose(c)
```

După executarea instrucţiunii `dispose(p)` valoarea variabilei de tip *referinţă* `p` este nedefinită.

Asupra variabilelor dinamice se pot efectua toate operaţiile admise de tipul de bază.

Exemplu:

```
Program P117;
 {Operaţii cu variabile dinamice }
 type AdresaInteger=^integer;
 var i, j, k : AdresaInteger;
     r, s, t : ^real;
 begin
  {crearea variabilelor dinamice de tipul integer }
  new(i); new(j); new(k);
  {operaţii cu variabilele create }
  i^:=1; j^:=2;
  k^:=i^+j^;
  writeln(k^);
  {crearea variabilelor dinamice de tipul real }
  new(r); new(s); new(t);
  {operaţii cu variabilele create }
  r^:=1.0; s^:=2.0;
  t^:=r^/s^;
  writeln(t^);
  {distrugerea variabilelor dinamice }
  dispose(i); dispose(j); dispose(k);
  dispose(r); dispose(s); dispose(t);
  readln;
 end.
```

Spre deosebire de variabilele statice, care ocupă zone de memorie stabilite de compilator, variabilele dinamice ocupă zone de memorie oferite de procedura `new`. Zonele respective sînt eliberate de procedura `dispose` şi pot fi reutilizate pentru crearea unor variabile dinamice noi. Prin urmare, procedurile `new` şi `dispose` asigură **alocarea (rezervarea) dinamică a memoriei**: spaţiul de memorie este atribuit unei variabile dinamice numai pe durata existenţei ei.

Numărul de variabile dinamice ce pot exista concomitent în timpul execuţiei unui program PASCAL depinde de tipul variabilelor şi spaţiul de memorie disponibil. În cazul în care tot spaţiul de memorie este deja ocupat, apelul procedurii `new` va declanşa o eroare de execuţie.

Exemplu:

```
Program P118;
 { Eroare: depăşirea capacităţii memoriei }
 label 1;
 var i : ^integer;
 begin
  1 : new(i);
      goto 1;
 end.
```

Alocarea dinamică a memoriei necesită o atenţie sporită din partea programatorului care este obligat să asigure crearea, distrugerea şi referirea corectă a variabilelor dinamice.

Întrebări şi exerciţii

❶ Care este diferenţa dintre variabilele statice şi variabilele dinamice?
❷ Cum se identifică variabilele dinamice?
❸ Indicaţi pe diagrama sintactică din *fig. 6.1* drumurile care corespund declaraţiilor de tipuri referinţă din programul `P117`.
❹ Se consideră declaraţiile:

```
type AdresaReal=^real;
var  r : AdresaReal;
```

Precizaţi mulţimea de valori ale tipului de date `AdresaReal` şi mulţimea de valori pe care le poate lua variabila dinamică `r^`.
❺ Ce operaţii se pot efectua cu valorile unui tip de date *referinţă*? Cu variabilele dinamice?
❻ Se consideră declaraţiile:

```
type AdresaTablou = ^array[1..10] of integer;
var  t : AdresaTablou.
```

Precizaţi mulţimea de valori ale tipului de date `AdresaTablou` şi mulţimea de valori pe care le poate lua variabila dinamică `t^`.
❼ Comentaţi programul:

```
Program P119;
 { Eroare }
 var r, s : ^real;
 begin
  r^:=1; s^:=2;
  writeln('r^=', r^, '  s^=', s^);
  readln;
 end.
```

❽ Elaboraţi un program în care se creează două variabile dinamice de tipul *şir de caractere*. Atribuiţi valori variabilelor create şi afişaţi la ecran rezultatul concatenării şirurilor respective.

❾ Ce va afişa pe ecran programul ce urmează?

```
Program P120;
 var i : ^integer;
 begin
  new(i); i^:=1;
  new(i); i^:=2;
  new(i); i^:=3;
  writeln(i^);
  readln;
 end.
```

❿ Comentaţi programul:

```
Program P121;
 { Eroare }
 var i, j : ^integer;
 begin
  new(i);
  i^:=1;
  dispose(i);
  new(j);
  j^:=2;
  dispose(j);
  writeln('i^=', i^, '  j^=', j^);
  readln;
 end.
```

⓫ Explicaţi expresia *alocarea dinamică a memoriei.*

6.2. STRUCTURI DE DATE

O **structură de date** este formată din datele propriu-zise şi relaţiile dintre ele. În funcţie de modul de organizare, o structură de date poate fi implicită sau explicită.

Tablourile, şirurile de caractere, articolele, fişierele şi mulţimile studiate în capitolele precedente sînt **structuri implicite de date**. Relaţiile dintre componentele acestor sructuri sînt predefinite şi nemodificabile. De exemplu, toate componentele unui şir de caractere au un nume comun, iar caracterul `s[i+1]` este succesorul caracterului `s[i]` în virtutea poziţiei ocupate.

Întrucît structura tablourilor, şirurilor de caractere, articolelor, mulţimilor şi fişierelor nu se modifică în timpul execuţiei oricărui pro-

gram sau subprogram, variabilele respective reprezintă **structuri statice de date**.

Folosind date cu structură implicită, putem rezolva reprezentativ o clasă limitată de probleme. În multe cazuri relaţiile dintre componente nu numai că se modifică dinamic, dar în acelaşi timp, pot deveni deosebit de complexe.

De exemplu, în cazul unui fir de aşteptare la o casă de bilete relaţiile dintre persoane se modifică: persoanele nou-sosite se aşază la rînd; persoanele în criză de timp pleacă fară să-şi mai procure bilete; persoanele care au plecat pentru un timp îşi păstrează rîndul ş.a.m.d. În cazul proiectării asistate de calculator a reţelelor de circulaţie, staţiile, rutele, capacitatea de trafic ş.a. pot fi stabilite interactiv de către utilizator. În astfel de situaţii utilizarea datelor cu structură implicită devine nenaturală, dificilă şi ineficientă.

Prin urmare, este necesară folosirea unor sructuri de date în care relaţiile dintre componente să fie reprezentate şi prelucrate în mod explicit. Acest efect se poate obţine ataşînd fiecărei componente o informaţie ce caracterizează relaţiile acesteia cu alte date ale structurii. În cele mai multe cazuri, informaţia suplimentară, numită **informaţie de structură**, se reprezintă prin variabilele de tipul referinţă.

Structurile de date componentele cărora sînt create şi eventual distruse în timpul execuţiei programului se numesc **structuri dinamice de date**. Structurile dinamice frecvent utilizate sînt: listele unidirecţionale, listele bidirecţionale, stivele, cozile, arborii ş. a.

O structură de date este **recursivă** dacă ea poate fi descompusă în date cu aceeaşi structură. Pentru exemplificare menţionăm listele unidirecţionale şi arborii care vor fi studiaţi în paragrafele următoare.

Întrebări şi exerciţii

❶ Explicaţi termenul *structură de date*. Daţi exemple.
❷ Care este diferenţa dintre sructurile implicite şi structurile explicite de date?
❸ O structură de date este **omogenă** dacă toate componentele sînt de acelaşi tip. În caz contrar structura de date este **eterogenă**. Daţi exemple de structuri omogene şi structuri eterogene de date.
❹ Care este diferenţa dintre sructurile statice şi structurile dinamice de date?
❺ Explicaţi termenul *structură recursivă de date*.

6.3. LISTE UNIDIRECŢIONALE

Listele unidirecţionale sînt structuri explicite şi dinamice de date formate din celule. Fiecare **celulă** este o variabilă dinamică de tipul `record` ce conţine, în principal, două cîmpuri: cîmpul datelor şi

cîmpul legăturilor. **Cîmpul datelor** memorează informaţia prelucrabilă asociată celulei. **Cîmpul legăturilor** furnizează indicatorul de adresă corespunzător celulei la care se poate ajunge din celula curentă. Se consideră că orice celulă poate fi atinsă pornind de la o celulă privilegiată, numită **baza listei**.

Pentru exemplificare, în *fig. 6.2* este prezentată o listă unidirecţională formată din 4 celule. Celulele conţin elementele A, B, C şi D.

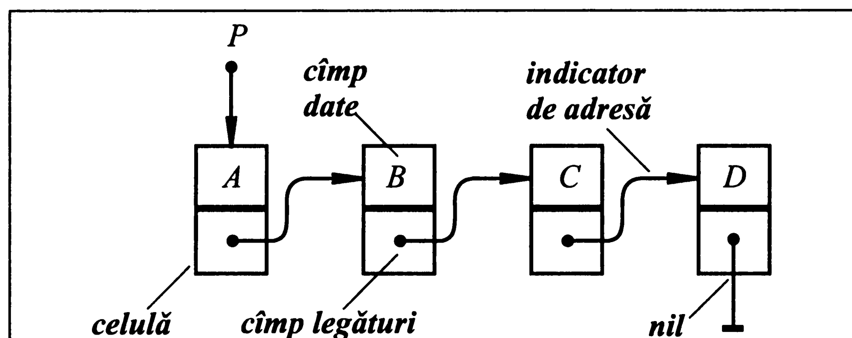


Fig. 6.2. Listă unidirecţională

Datele necesare pentru crearea şi prelucrarea unei liste unidirecţionale pot fi definite prin declaraţii de forma:

```
type  AdresaCelula=^Celula;
      Celula=record
                Info : string;
                Urm  : AdresaCelula;
                end;
var   P : AdresaCelula;
```

Informaţia utilă asociată unei celule se memorează în cîmpul `Info`, iar adresa celulei următoare în cîmpul `Urm`. Pentru simplificare se consideră că cîmpul `Info` este de tipul şir de caractere. Ultima celulă din listă va avea în cîmpul `Urm` valoarea **`nil`**. Adresa primei celule (adresa de bază) este memorată în variabila de tip referinţă `P` (*fig. 6.2*).

Subliniem faptul că în definiţia tipului referinţă `AdresaCelula` tipul de bază `Celula` încă nu este cunoscut. Conform regulilor limbajului PASCAL, acest lucru este posibil numai în cazul variabilelor dinamice, cu condiţia ca tipul de bază să fie definit mai tîrziu în aceeaşi declaraţie.

O listă unidirecţională poate fi creată adăugînd la vîrful listei cîte un element. Iniţial lista în curs de construcţie este vidă, adică nu conţine nici o celulă.

Exemplu:

```
Program P122;
 { Crearea listei unidirecţionale A->B->C->D }
 type AdresaCelula=^Celula;
      Celula=record
                     Info : string;
                      Urm : AdresaCelula;
              end;
 var P,          { adresa de bază }
     R, V : AdresaCelula;
 begin
  { 1 - iniţial lista este vidă }
  P:=nil;
  { 2 - adăugarea celulei A }
  new(R);        { crearea unei celule }
  P:=R;          { iniţializarea adresei de bază }
  R^.Info:='A'; { încărcarea informaţiei utile }
  R^.Urm:=nil;  { înscrierea indicatorului
                  "sfîrşit de listă" }
  V:=R;          { memorarea adresei vîrfului }
  { 3 - adăugarea celulei B }
  new(R);        { crearea unei celule }
  R^.Info:='B'; { încărcarea informaţiei utile }
  R^.Urm:=nil;  { înscrierea indicatorului
                   "sfîrşit de listă" }
  V^.Urm:=R;     { crearea legăturii A -> B }
  V:=R;          { actualizarea adresei vîrfului }
  { 4 - adăugarea celulei C }
  new(R);        { crearea unei celule }
  R^.Info:='C'; { încărcarea informaţiei utile }
  R^.Urm:=nil;  { înscrierea indicatorului
                   "sfîrşit de listă" }
  V^.Urm:=R;     { crearea legăturii B -> C }
  V:=R;          { actualizarea adresei vîrfului }
  { 5 - adăugarea celulei D }
  new(R);        { crearea unei celule }
  R^.Info:='D'; { încărcarea informaţiei utile }
  R^.Urm:=nil;  { înscrierea indicatorului
                   "sfîrşit de listă" }
  V^.Urm:=R;     { crearea legăturii C -> D }
  V:=R;          { actualizarea adresei vîrfului }
  { afişarea listei create }
  R:=P;
  while R<>nil do begin
                    writeln(R^.Info);
                    R:=R^.Urm
                  end;
  readln;
 end.
```

Procesul de construire a listei în studiu este prezentat în *fig. 6.3*. Variabila `V` (adresa vîrfului) din programul `P122` reține adresa ultimei celule deja create pentru a-i poziționa indicatorul de adresă

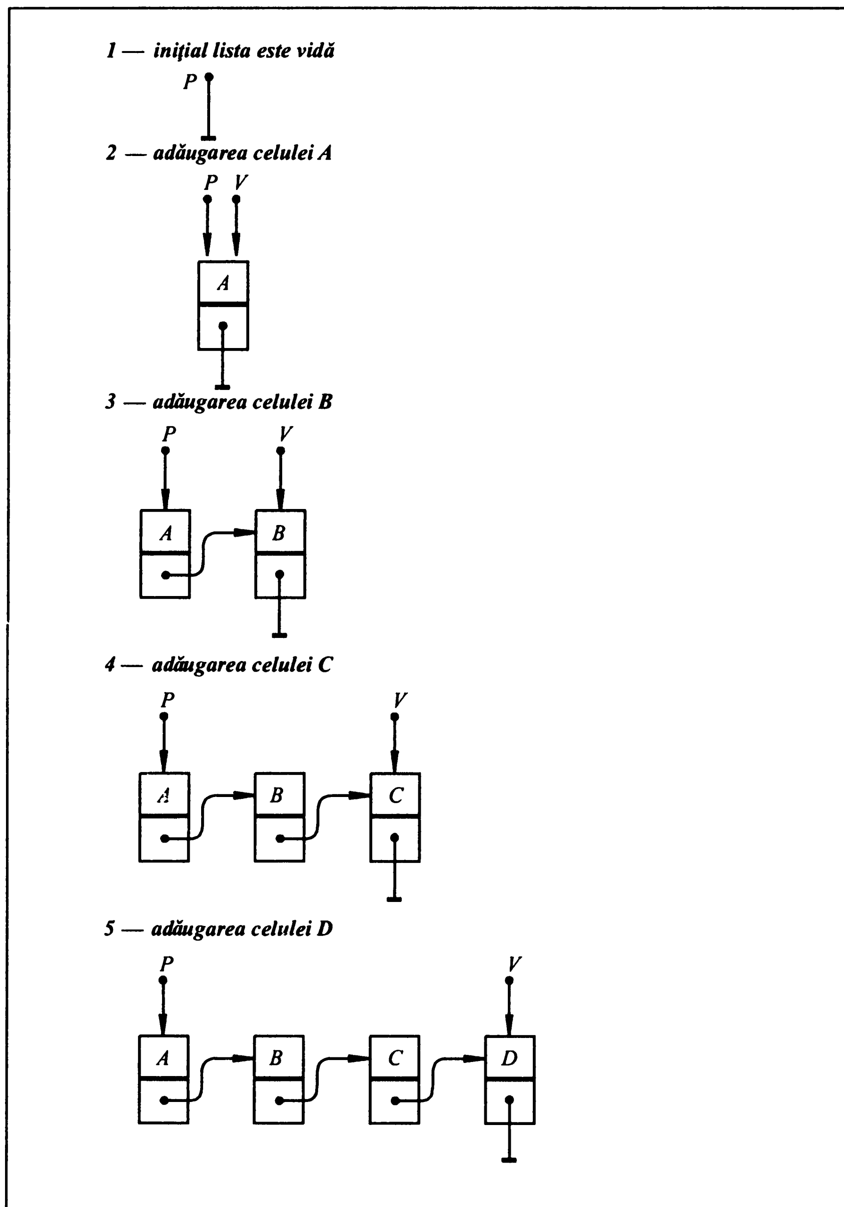


Fig. 6.3. Crearea listei unidirecționale

V^.Urm. Se procedează astfel pentru că în momentul în care completăm cîmpurile R^.Info şi R^.Urm ale celulei curente încă nu se cunoaşte adresa celulei ce urmează.

Listele cu un număr arbitrar de celule pot fi create şi afişate cu ajutorul procedurilor respective din programul P123:

```
Program P123;
{ Crearea listelor unidirectionale}
type AdresaCelula=^Celula;
     Celula=record
              Info : string;
               Urm : AdresaCelula;
             end;
var p,q,r : AdresaCelula;
    s : string;
    i: integer;
procedure Creare;
begin
 p:=nil;
 write('s=' ); readln(s);
 new(r); r^.Info:=s; r^.Urm:=nil;
 p:=r; q:=r;
 write('s=' );
  while not eof do
   begin
     readln(s);write('s=' );
     new(r); r^.Info:=s; r^.Urm:=nil;
     q^.Urm:=r; q:=r
   end;
end; { Creare }
procedure Afisare;
begin
 r:=p;
 while r<>nil do
   begin
     writeln(r^.Info);
     r:=r^.Urm;
   end;
readln;
end; { Afisare }

begin
 Creare;
 Afisare;
end.
```

Orice listă unidirecţională poate fi definită **recursiv** după cum urmează:

a) o celulă este o listă unidirecţională;

b) o celulă ce conţine o legătură către o altă listă unidirecţională este o listă unidirecţională.

Pentru a sublinia faptul că listele unidirecţionale sînt structuri recursive de date, declaraţiile respective pot fi transcrise în forma:

```
type  Lista=^Celula;
      Celula=record
               Info : string;
               Urm  : Lista
             end;
var   P : Lista;
```

Proprietăţile listelor unidirecţionale pot fi reproduse parţial prin memorarea elementelor respective într-un tablou unidimensional. De exemplu, datele din *fig. 6.2* pot fi reprezentate în forma:

```
var L : array [1..4] of string;
  ...
  L[1]:='A';
  L[2]:='B';
  L[3]:='C';
  L[4]:='D'
  ...
```

Însă o astfel de reprezentare nu permite crearea şi prelucrarea structurilor de date cu un număr arbitrar de elemente.

Întrebări şi exerciţii

❶ Cum se definesc datele necesare pentru crearea unei liste?
❷ Care este destinaţia cîmpului datelor din componenţa unei celule? Care este destinaţia cîmpului legăturilor? Ce informaţie se înscrie în acest cîmp?
❸ Scrieţi un program care creează lista unidirecţională din *fig. 6.2*, adăugînd cîte un element la baza listei.
❹ Elaboraţi o procedură care schimbă cu locul două elemente din listă.
❺ De la tastatură se citesc numere întregi diferite de zero. Se cere să se creeze două liste, una a numerelor negative, iar alta — a numerelor pozitive.
❻ Prin ce se explică faptul că listele unidirecţionale sînt structuri recursive de date?

6.4. PRELUCRAREA LISTELOR UNIDIRECŢIONALE

Operaţiile frecvent utilizate în cazul listelor unidirecţionale sînt:
– parcurgerea listei şi prelucrarea informaţiei utile asociate fiecărei celule;
– căutarea unui anumit element, identificat prin valoarea sa;
– includerea (inserarea) unui element într-un anumit loc din listă;
– excluderea (ştergerea) unui element dintr-o listă ş.a.

Presupunem că există o listă nevidă (*fig. 6.2*) definită prin declarațiile:

```
type  AdresaCelula=^Celula;
      Celula=record;
               Info : string;
               Urm  : AdresaCelula
              end;
```

Variabila P indică adresa de bază a listei în studiu.

Parcurgerea listei se realizează conform următoarei secvențe de instrucțiuni:

```
R:=P; {poziționare pe celula de bază }
while R<>nil do
begin
  {prelucrarea informației din cîmpul R^.Info }
  R:=R^. Urm; { poziționare pe celula următoare }
end
```

Căutarea celulei ce conține elementul specificat de variabila Cheie este realizată de secvența:

```
R:=P;
while R^.Info<>Cheie do R:=R^.Urm
```

Adresa celulei în studiu va fi reținută în variabila R.

Subliniem faptul că această secvență se execută corect numai în cazul în care lista include cel puțin o celulă ce conține informația căutată. În caz contrar se ajunge la vîrful listei, variabila R ia valoarea **nil**, iar dereperarea R^ provoacă o eroare de execuție. Pentru a evita astfel de erori, se utilizează secvența:

```
  R:=P;
  while R<>nil do
  begin
    if R^.Info=Cheie then goto 1;
    R:=R^.Urm
  end;
1:...
```

Întrucît listele unidirecționale sînt structuri recursive de date, operația de căutare poate fi realizată și de un subprogram recursiv:

```
type  Lista=^AdresaCelula;
      Celula=record;
               Info : string;
               Urm : Lista;
              end;
var   P : Lista;
...
function Caut(P : Lista; Cheie : string):Lista;
```

```
begin
  if P=nil then
  Caut:=nil
  else
    if P^.Info=Cheie then
    Caut:=P
    else Caut:=Caut(P^.Urm, Cheie)
 end;
```

Funcţia `Caut` returnează adresa de bază a sublistei ce conţine în prima celulă, dacă există, elementul specificat de parametrul `Cheie`.

Includerea celulei referite de indicatorul `Q` după celula referită de indicatorul `R` (*fig. 6.4*) este realizată de secvenţa de instrucţiuni:

```
Q^.Urm:=R^.urm;
R^.Urm:=Q
```

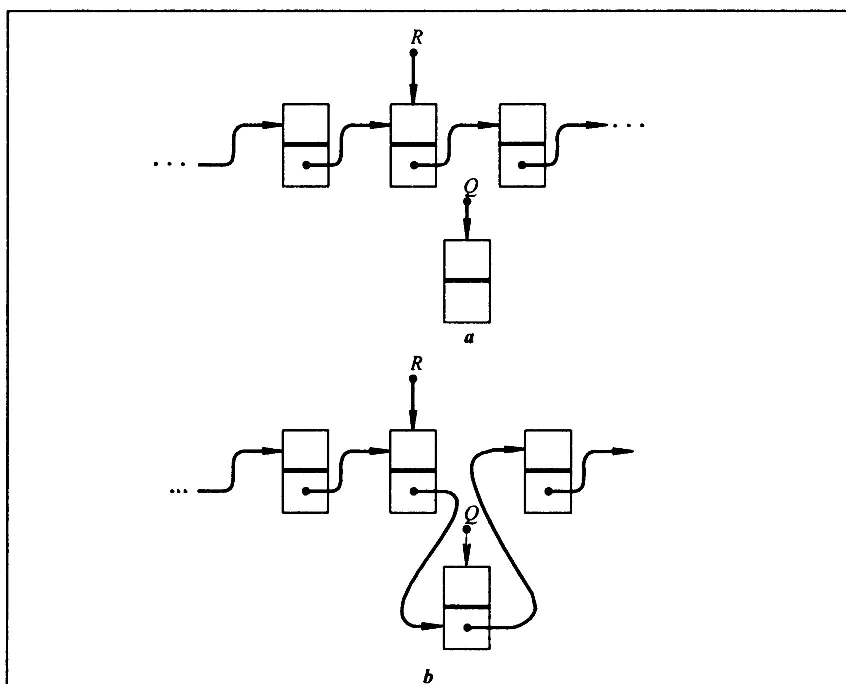


Fig. 6.4. Includerea unui element în listă:
a – lista pînă la includere; *b* – lista după includere

Excluderea celulei `R` din listă necesită aflarea adresei `Q` a celulei precedente şi modificarea indicatorului de adresă `Q^.Urm` (*fig. 6.5*):

```
Q:=P;
while Q^.Urm<>R do Q:=Q^.Urm;
Q^.Urm:=R^.Urm;
```

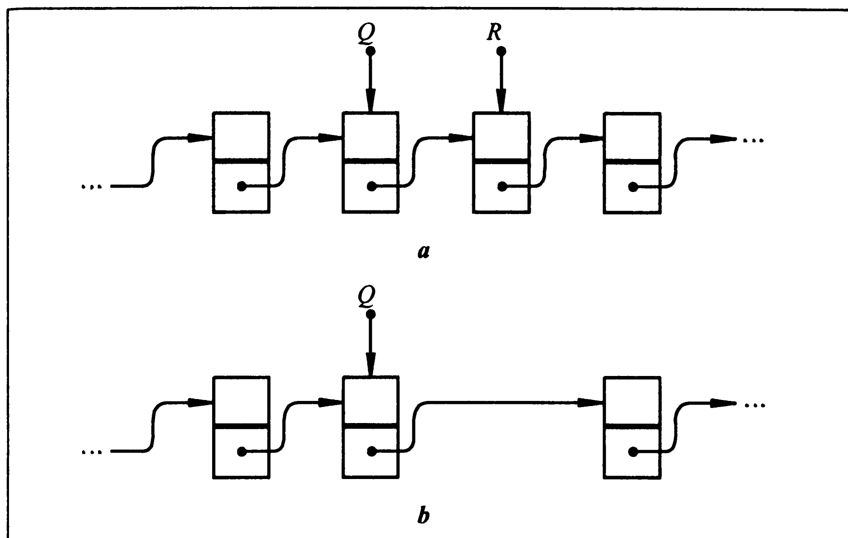


Fig. 6.5. Excluderea unui element din listă:
a – lista pînă la excludere; b – lista după excludere

Menţionăm că includerea sau excluderea elementului din baza listei necesită actualizarea adresei de bază P.

Exemplu:

```
Program P124;
 {Crearea şi prelucrarea unei liste unidirecţionale }
 type AdresaCelula=^Celula;
      Celula=record
                  Info : string;
                  Urm : AdresaCelula;
                end;
 var P : AdresaCelula; {adresa de bază }
     c : char;
 procedure Creare;
 var R, V : AdresaCelula;
 begin
if P<>nil then writeln('Lista există deja')
    else begin
           writeln('Daţi lista:');
           while not eof do
             begin
               new(R);
               readln(R^.Info);
               R^.Urm:=nil;
               if P=nil then begin P:=R; V:=R end
                      else begin V^.Urm:=R; V:=R
```

```
        end;
          end;
              end;
end; { Creare }

 procedure Afis;
var R : AdresaCelula;
begin
 if P=nil then writeln('Lista este vidă')
         else begin
                writeln('Lista curentă:');
                R:=P;
                while R<>nil do
               begin
                  writeln(R^.Info);
                   R:=R^.Urm;
                end;
              end;
 readln;
end; { Afis }
procedure Includ;
label 1;
var Q, R : AdresaCelula;
   Cheie : string;
begin
  new(Q);
  write('Daţi elementul ce urmează');
  writeln(' să fie inclus:');
  readln(Q^.Info);
  write('Indicaţi elementul după care');
  writeln(' se va face includerea:');
  readln(Cheie);
  R:=P;
  while R<>nil do
    begin
      if R^.Info=Cheie then goto 1;
      R:=R^.Urm;
    end;
1:if R=nil then begin
                 writeln('Element inexistent');
                dispose(Q);
               end;
          else begin
                 Q^.Urm:=R^.Urm;
                 R^.Urm:=Q;
               end;
end; {  Includ }
procedure Exclud;
label 1;
var Q, R : AdresaCelula;
```

```
   Cheie : string;
begin
  write('Daţi elementul ce urmează');
  writeln(' să fie exclus:');
  readln(Cheie);
  R:=P;
  Q:=R;
  while R<>nil do
    begin
      if R^.Info=Cheie then goto 1;
      Q:=R;
      R:=R^.Urm;
    end;
1:if R=nil then writeln('Element inexistent')
          else begin
                 if R=P then P:=R^.Urm
                else Q^.Urm:=R^.Urm;
                 dispose(R);
               end;
end; { Exclud }
begin
 P:=nil; { iniţial lista este vidă }
 repeat
   writeln('Meniu:');
   writeln('C - Crearea listei');
   writeln('I - Includerea elementului');
   writeln('E - Excluderea elementului');
   writeln('A - Afişarea listei la ecran');
   writeln('O - Oprirea programului');
   write('Opţiunea='); readln(c);
   case c of
       'C' : Creare;
       'I' : Includ;
       'E' : Exclud;
       'A' : Afis;
       'O' :
       else writeln('Opţiune necunoscută')
   end;
 until c='O';
end.
```

Procedura `Creare` formează o listă unidirecţională cu un număr arbitrar de celule. Informaţia utilă asociată fiecărei celule se citeşte de la tastatură. Procedura `Afis` afişează elementele listei la ecran. Procedura `Includ` citeşte de la tastatură elementul ce urmează să fie inclus şi elementul după care se va face includerea. În continuare se caută celula ce conţine elementul specificat, după care, dacă există, este inclusă celula nou creată. Procedura `Exclud` caută şi elimină, dacă există, celula ce conţine elementul citit de la tastatură.

Întrebări şi exerciţii

❶ Scrieţi o funcţie nerecursivă care returnează adresa vîrfului listei unidirecţionale. Transcrieţi această funcţie într-o formă recursivă.

❷ Transcrieţi procedurile `Includ` şi `Exclud` din programul `P124`, fără a utiliza instrucţiunea **goto**.

❸ Se consideră următoarele tipuri de date:

```
type  AdresaCandidat=^Candidat;
      Candidat=record
                 NumePrenume : string;
                 NotaMedie : real;
                 Urm : AdresaCandidat
               end;
```

Elaboraţi un program care:

a) creează o listă unidirecţională cu componente de tipul `Candidat`;
b) afişează lista pe ecran;
c) exclude din listă candidatul care îşi retrage actele;
d) include în listă candidatul care depune actele;
e) afişează pe ecran candidaţii cu media mai mare de 7,5;
f) creează o listă suplimentară formată din candidaţii cu media mai mare de 9,0;
g) exclude din listă toţi candidaţii cu media mai mică de 6,0.

❹ Elaboraţi o procedură care:

a) reordonează elementele listei unidirecţionale conform unui anumit criteriu;
b) concatenează două liste unidirecţionale;
c) descompune o listă în două liste;
d) selectează din listă elementele care corespund unui anumit criteriu.

❺ Elementele listei unidirecţionale sînt memorate într-un tablou unidirecţional. Elaboraţi procedurile necesare pentru:

a) parcurgerea listei;
b) căutarea unui anumit element;
c) includerea unui element;
d) excluderea unui element.

Care sînt avantajele şi neajunsurile acestei reprezentări? Se consideră că lista va conţine cel mult 100 de elemente.

❻ Scrieţi o funcţie recursivă ce returnează numărul de celule dintr-o listă unidirecţională.

❼ Scrieţi un subprogram recursiv care exclude o anumită celulă din listă.

❽ Prin **cuvînt** se înţelege orice secvenţă nevidă formată din literele alfabetului latin. Elaboraţi un program care formează lista cuvintelor întîlnite într-un fişier *text* şi calculează numărul de apariţii al fiecărui cuvînt. Examinaţi cazurile:

a) cuvintele se includ în listă în ordinea primei lor apariţii în text;
b) cuvintele se includ în listă în ordine alfabetică.

Se consideră că literele mari şi mici sînt identice.

6.5. LISTE BIDIRECŢIONALE

Fiecare celulă a unei liste bidirecţionale conţine în cîmpul legăturilor doi indicatori de adresă. Primul indicator identifică celula următoare, iar al doilea — celula precedentă. Pentru exemplificare, în *fig. 6.6* este prezentată o listă bidirecţională formată din 4 celule ce conţin elementele A, B, C şi D.

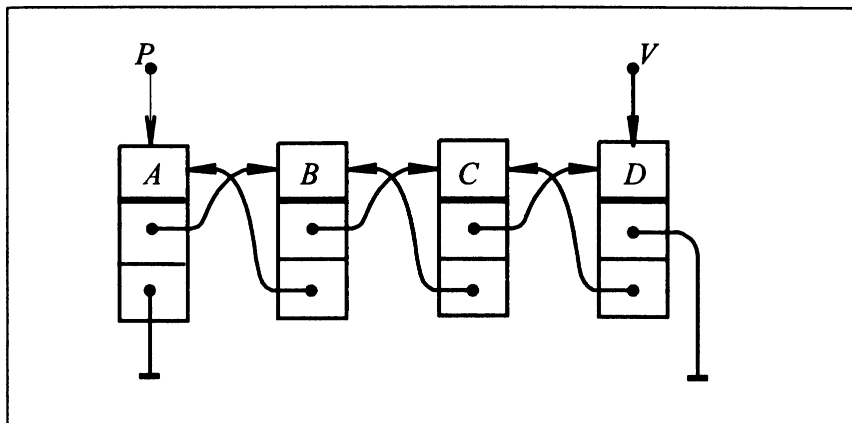


Fig. 6.6. Listă bidirecţională

Datele necesare pentru crearea şi prelucrarea unei liste bidirecţionale pot fi definite prin declaraţii de forma:

```
type  AdresaCelula=^Celula;
      celula=record
               Info : string;
               Urm, Prec : AdresaCelula;
             end;
var   P, V : AdresaCelula;
```

Variabilele de tip *referinţă* P şi V memorează respectiv adresa primei celule (adresa de bază) şi adresa ultimei celule (adresa vîrfului).

O listă bidirecţională poate fi creată, adăugînd la vîrful listei cîte un element. Iniţial lista în curs de construcţie este vidă.

Exemplu:

```
Program P125;
 { Crearea listei bidirecţionale A<->B<->C<->D }
 type AdresaCelula=^Celula;
      Celula=record
                 Info  : string;
              Urm, Prec : AdresaCelula;
             end;
```

```
var P,          { adresa de bază }
    V,          { adresa vîrfului }
    R : AdresaCelula;
begin
 { 1 - inițial lista este vidă }
 P:=nil; V:=nil;
 { 2 - adăugarea celulei A }
 new(R);
 P:=R;          { inițializarea adresei de bază }
 R^.Info:='A';
 R^.Prec:=nil;
 R^.Urm:=nil;
 V:=R;          { inițializarea adresei vîrfului }
 { 3 - adăugarea celulei B }
 new(R);
 R^.Info:='B';
 R^.Prec:=V;    { crearea legăturii B -> A }
 R^.Urm:=nil;
 V^.Urm:=R;     { crearea legăturii A -> B }
 V:=R;          { actualizarea adresei vîrfului }
 { 4 - adăugarea celulei C }
 new(R);
 R^.Info:=' C';
 R^.Prec:=V;    { crearea legăturii C -> B }
 R^.Urm:=nil;
 V^.Urm:=R;     { crearea legăturii B -> C }
 V:=R;          { actualizarea adresei vîrfului }
 { 5 - adăugarea celulei D }
 new(R);
 R^.Info:='D';
 R^.Prec:=V;    { crearea legăturii D -> C}
 R^.Urm:=nil;
 V^.Urm:=R;      {  crearea legăturii C -> D }
 V:=R;           {  actualizarea adresei vîrfului }

 { afişarea listei create }
 R:=P;
 while R<>nil do begin
                  writeln(R^.Info);
                  R:=R^.Urm
                end;
 readln;
end.
```

Procesul de construire a listei în studiu este prezentat în *fig. 6.7*.
Asupra listelor bidirecționale se efectuează următoarele operații:

– parcurgerea listei de la bază la vîrf;
– parcurgerea listei de la vîrf la bază;
– căutarea unui element, identificat prin valoarea sa;
– includerea unui element în listă;

– excluderea unui element din listă;
– reordonarea elementelor listei conform unui anumit criteriu;
– concatenarea listelor;
– descompunerea unei liste în mai multe liste ş.a.

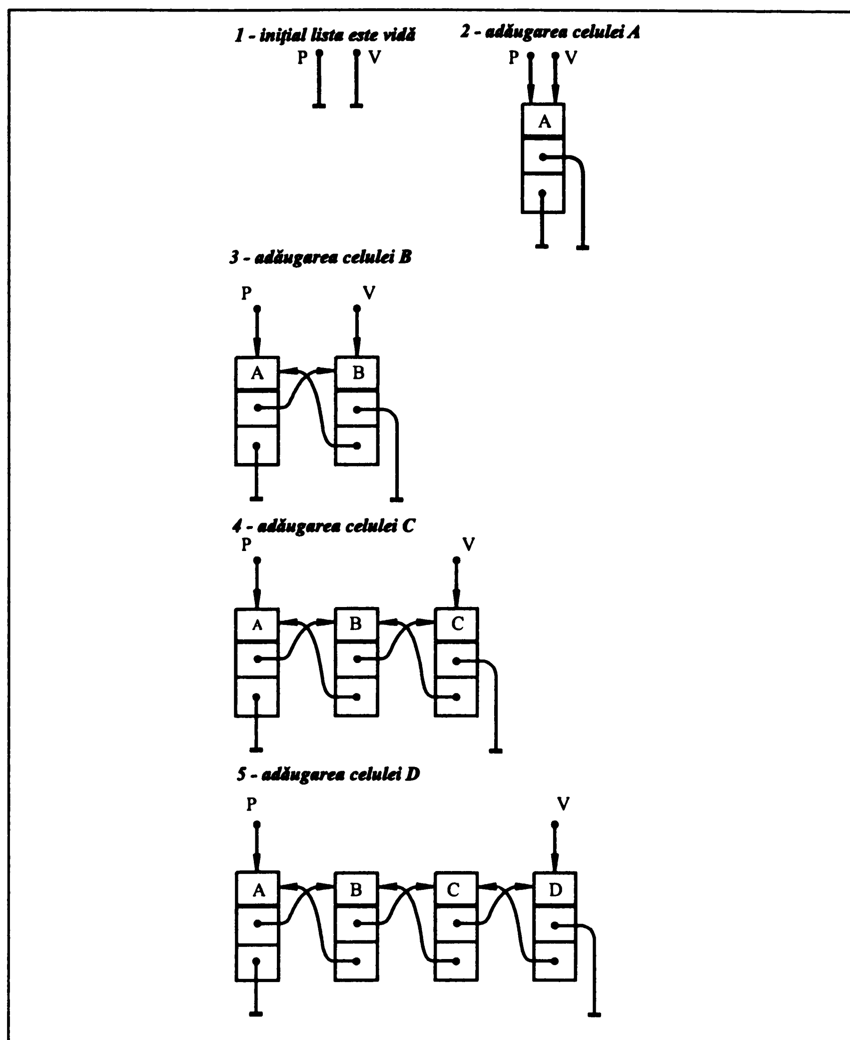


Fig. 6.7. Crearea listei bidirecţionale

Ca şi în cazul listelor unidirecţionale, includerea şi excluderea elementelor, concatenarea şi descompunerea listelor constă în crearea şi eventual distrugerea celulelor sau a legăturilor respective. Evident, parcurgerea listelor bidirecţionale în ambele direcţii se realizează mai

simplu datorită faptului că pentru fiecare celulă din listă se cunoaşte nu numai adresa celulei următoare, dar şi adresa celulei precedente.

Exemplu:

```
Program P126;
 { Crearea şi parcurgerea unei liste bidirecţionale }
 type AdresaCelula=^Celula;
      Celula=record
               Info : string;
               Urm, Prec : AdresaCelula;
             end;
 var P,                    { adresa de bază }
     V : AdresaCelula;     { adresa vîrfului }
 procedure Creare;
 var R : AdresaCelula;
 begin
   P:=nil; V:=nil;
   writeln('Daţi lista:');
   while not eof do
     begin
       new(R);
       readln(R^.Info);
       R^.Prec:=nil;
       R^.Urm:=nil;
       if P=nil then begin P:=R; V:=R end
                else begin V^.Urm:=R; R^.Prec:=V;
V:=R end;
       end;
 end; { Creare }

 procedure Parcurgere;
 var R : AdresaCelula;
 begin
   writeln('Parcurgere de la bază la vîrf:');
   R:=P;
   while R<>nil do
     begin
       writeln(R^.Info);
       R:=R^.Urm;
     end;
   writeln('Parcurgere de la vîrf la bază:');
   R:=V;
   while R<>nil do
     begin
       writeln(R^.Info);
       R:=R^.Prec;
     end;
   readln;
 end; { Parcurgere }
```

```
begin
  Creare;
  Parcurgere;
end.
```

Întrebări şi exerciţii

❶ Cum se definesc datele necesare pentru crearea unei liste bidirecţionale? Care este destinaţia cîmpurilor `Prec` şi `Urm` ale unei celule?

❷ Scrieţi un program care creează lista bidirecţională din *fig. 6.6*, adăugînd cîte un element la baza listei.

❸ Completaţi programul `P126` cu procedurile necesare pentru includerea şi excluderea elementelor din lista bidirecţională creată.

❹ Elaboraţi o procedură care:

a) transformă o listă unidirecţională într-o listă bidirecţională;
b) transformă o listă bidirecţională într-o listă unidirecţională.

Sînt oare posibile aceste transformări fără a dubla volumul de memorie alocat listei iniţiale?

❺ Scrieţi o procedură care parcurge o listă unidirecţională de la vîrf la bază. Comparaţi complexitatea acestei proceduri cu complexitatea secvenţei respective de instrucţiuni din programul `P126`.

❻ Care sînt avantajele şi neajunsurile listelor bidirecţionale comparativ cu listele unidirecţionale? Comparativ cu tablourile unidimensionale?

❼ În jurul arbitrului sînt aranjaţi în cerc *N* jucători. Începînd cu jucătorul *m*, cu raţia *k*, jucătorii sînt eliminaţi. Elaboraţi un program care afişează pe ecran ordinea de ieşire din cerc.
Indicaţie: Jucătorii aranjaţi în cerc pot fi reprezentaţi printr-o listă bidirecţională în care sînt stabilite legături între vîrf şi bază (listă inelară).

6.6. STIVA

Prin **stivă** (în limba engleză *stack*) înţelegem o listă unidirecţională cu proprietatea că operaţiile de introducere şi extragere a elementelor se fac la un singur capăt al ei. Poziţia ocupată în stivă de ultimul element introdus poartă numele de **vîrf**. O stivă fără nici un element se numeşte **stivă vidă**.

Pentru exemplificare, în *fig. 6.8* este prezentată o stivă care conţine elementele A, B, C.

Datele necesare pentru crearea şi prelucrarea unei stive pot fi definite prin declaraţii de forma:

```
type  AdresaCelula=^Celula;
      Celula=record
               Info : string;
               Prec : AdresaCelula
      end;
var   S : AdresaCelula;
```

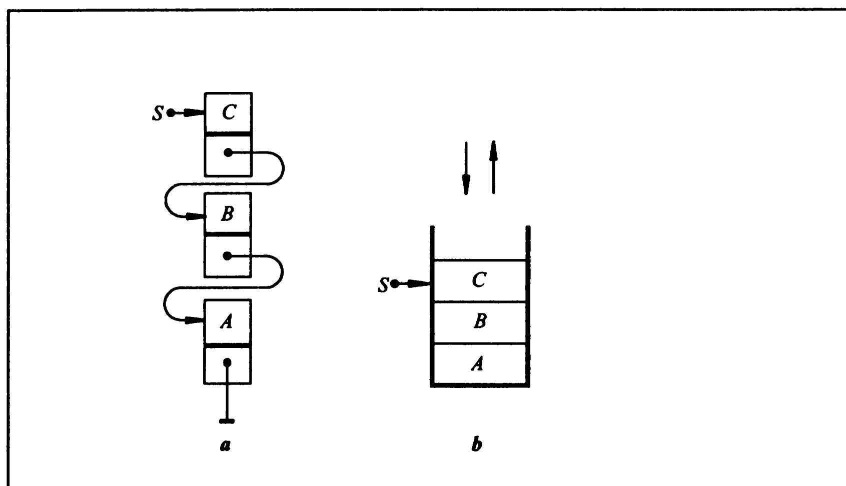


Fig. 6.8. Stiva:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

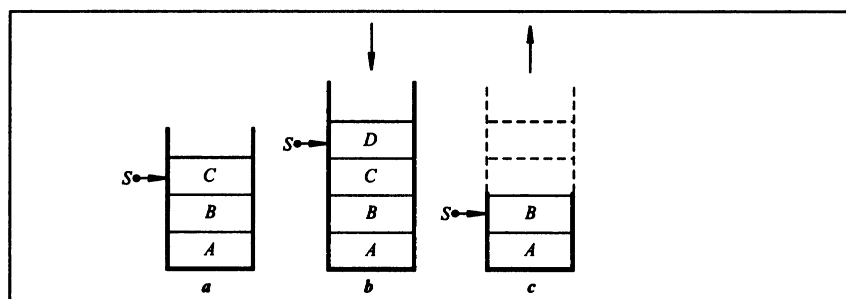


Fig. 6.9. Introducerea şi extragerea elementelor din stivă:
a – stiva iniţială; *b* – introducerea elementului D;
c – extragerea elementelor D, C

Adresa vîrfului stivei este memorată în variabila de tip *referinţă* `S`. Adresa celulei precedente din stivă este memorată în cîmpul `Prec`.

Operaţia de introducere a unui element în stivă (*fig. 6.9*) este efectuată de secvenţa de instrucţiuni:

```
new(R); { crearea unei celule }
{ încărcarea informaţiei utile
  în cîmpul R^.Info }
R^.Prec:=S; { crearea legăturii
              către celula precedentă din stivă }
S:=R; { actualizarea adresei vîrfului }
```

unde `R` este o variabilă de tipul `AdresaCelula`.

Extragerea unui element din stivă (*fig. 6.9*) este efectuată de secvenţa:

```
R:=S; { memorarea adresei celulei extrase }
 { prelucrarea informaţiei din cîmpul R^.Info }
 S:=S^.Prec; { eliminarea celulei din stivă }
 dispose(R); { distrugerea celulei extrase }
```

Exemplu:

```
Program P127;
 {Crearea şi prelucrarea unei stive }
 type AdresaCelula=^Celula;
      Celula=record
                   Info : string;
                   Prec : AdresaCelula;
             end;
 var S : AdresaCelula; {adresa vîrfului }
     c : char;
 procedure Introduc;
 var R : AdresaCelula;
 begin
  new(R);
  write('Daţi elementul ce urmează');
  writeln(' să fie introdus:');
  readln(R^.Info);
  R^.Prec:=S;
  S:=R;
 end; { Includ }

 procedure Extrag;
 var R : AdresaCelula;
 begin
  if S=nil then writeln('Stiva este vidă')
           else begin
                 R:=S;
                 write('Este extras');
                 writeln(' elementul:');
                 writeln(R^.Info);
                 S:=S^.Prec;
                 dispose(R);
                end;
 end; { Extrag }

 procedure Afis;
 var R : AdresaCelula;
 begin
  if S=nil then writeln('Stiva este vidă')
    else begin
           writeln('Stiva include elementele:');
           R:=S;
```

```
            while R<>nil do begin
                                writeln(R^.Info);
                                R:=R^.Prec;
                              end;
          end;
  readln;
 end; { Afis }
 begin
 S:=nil;  { iniţial stiva este vidă }
 repeat
   writeln('Meniu:');
   writeln('I - Introducerea elementului;');
   writeln('E - Extragerea elementului');
   writeln('A - Afişarea stivei pe ecran');
   writeln('O - Oprirea programului');
   write('Opţiunea='); readln(c);
   case c of
       'I' : Introduc;
       'E' : Extrag;
       'A' : Afis;
       'O' :
       else writeln('Opţiune necunoscută')
   end;
 until c='O';
end.
```

Stivele mai poartă şi numele de **liste** ***LIFO*** (*last in, first out* — ultimul element care a intrat în stivă va fi primul care va ieşi din ea) şi sînt frecvent utilizate pentru alocarea dinamică a memoriei în cazul procedurilor şi funcţiilor recursive. Evident, stivele pot fi simulate utilizînd tablourile unidimensionale **array**[1..*n*] **of** ..., însă o astfel de reprezentare este limitată de cele *n* componente ale tablourilor.

Întrebări şi exerciţii

❶ Care este ordinea de introducere şi extragere a datelor din stivă?

❷ De la tastatură se citesc mai multe numere naturale. Afişaţi numerele în studiu pe ecran în ordinea inversă citirii.

❸ În *fig. 6.10* este reprezentată schema de manevrare a vagoanelor de tren într-un depou.
Elaboraţi un program care citeşte de la tastatură şi afişează pe ecran datele despre fiecare vagon intrat sau ieşit din depou. Datele în studiu includ:

- numărul de înmatriculare (integer);
- staţia de înmatriculare (string);
- anul fabricării (1960..2000);
- tipul vagonului (string);
- proprietarul vagonului (string);
- capacitatea de încărcare (real).

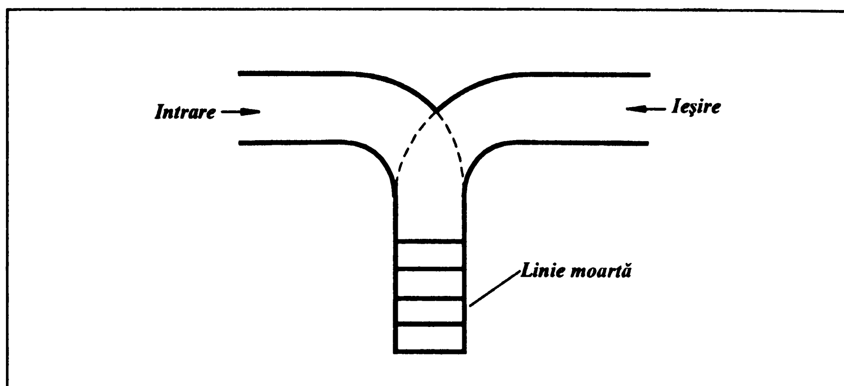


Fig. 6.10. Schema de manevrare a vagoanelor de tren

❹ Pot fi oare folosite listele bidirecţionale pentru crearea unei stive?

❺ Colectivele temporare de muncă sînt formate şi desfiinţate în ordinea "ultimul angajat va fi primul care va fi concediat". Elaboraţi un program care citeşte de la tastatură şi afişează pe ecran datele despre fiecare persoană angajată sau concediată. Datele în studiu includ :

- numele (`string`);
- prenumele (`string`);
- anul naşterii (`1930..1985`);
- data angajării (ziua, luna, anul).

❻ Se consideră şiruri finite de caractere formate din parantezele (,), [,], {, }. Un şir este **corect** numai atunci cînd el poate fi construit cu ajutorul următoarelor reguli:

a) şirul vid este corect;
b) dacă *A* este un şir corect, atunci (*A*), [*A*] şi {*A*} sînt şiruri corecte;
c) dacă *A* şi *B* sînt şiruri corecte, atunci *AB* este un şir corect.

De exemplu, şirurile ( ), [ ], { }, [( )], ((({[ ]}))([ ])) sînt corecte, iar şirurile ([, ( )[ ]{{, ([)] nu sînt corecte. Elaboraţi un program care verifică dacă şirul citit de la tastatură este corect.

Indicaţie. Problema poate fi rezolvată printr-o singură parcurgere a şirului supus verificării. Dacă caracterul curent este (, [sau {, el este depus în stivă. Dacă vîrful stivei şi caracterul curent formează una din perechile ( ), [ ] sau { }, paranteza respectivă este scoasă din stivă. În cazul unui şir corect, după examinarea ultimului caracter din şir stiva rămîne vidă.

❼ Elementele stivei sînt memorate într-un tablou unidimensional. Elaboraţi procedurile necesare pentru introducerea şi extragerea elementelor din stivă. Care sînt avantajele şi neajunsurile acestei reprezentări? Se consideră că stiva conţine cel mult 100 de elemente.

6.7. COZI

Prin **coadă** (în engleză *queue*) înţelegem o listă unidirecţională în care toate introducerile se efectuează la unul din capete, iar extragerele se efectuează la celălalt capăt. O coadă fără nici un element se numeşte **coadă vidă**.

Pentru exemplificare, în *fig. 6.11* este prezentată o coadă care conţine elementele A, B, C.

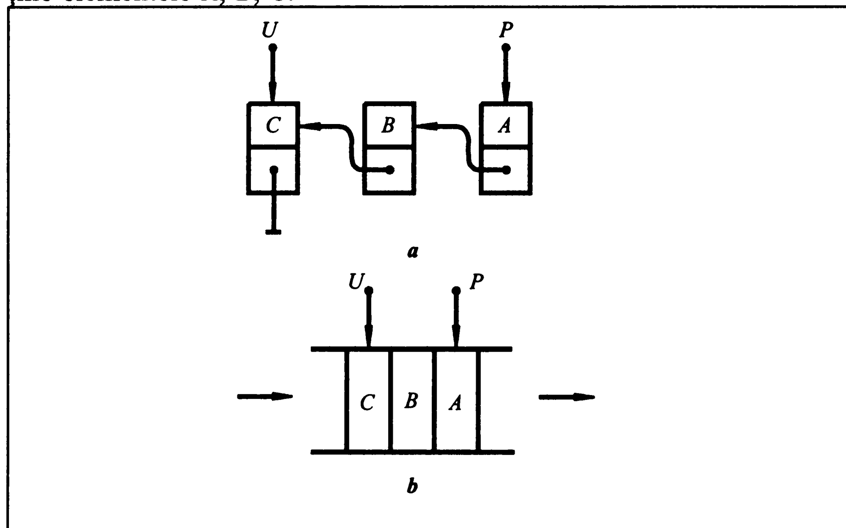


Fig. 6.11. Coada:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

Datele necesare pentru crearea şi prelucrarea unei cozi pot fi definite prin declaraţii de forma:

```
type  AdresaCelula=^Celula;
      Celula=record
               Info : string;
               Urm : AdresaCelula
             end;
var   P, U : AdresaCelula;
```

Adresa primului element din coadă este memorată în variabila de tip referinţă P, iar adresa ultimului element în variabila U. Adresa celulei următoare din coadă este memorată în cîmpul Urm.

Operaţia de introducere a unui element (*fig. 6.12*) este efectuată de secvenţa de instrucţiuni:

```
new (R); { crearea unei celule }
{ încărcarea informaţiei utile
```

```
   în cîmpul R^.Info }
R^.Urm:=nil; { înscrierea indicatorului
                 "ultimul element" }
U^.Urm:=R; { adăugarea celulei la coadă }
U:=R; { actualizarea adresei ultimei celule }
```

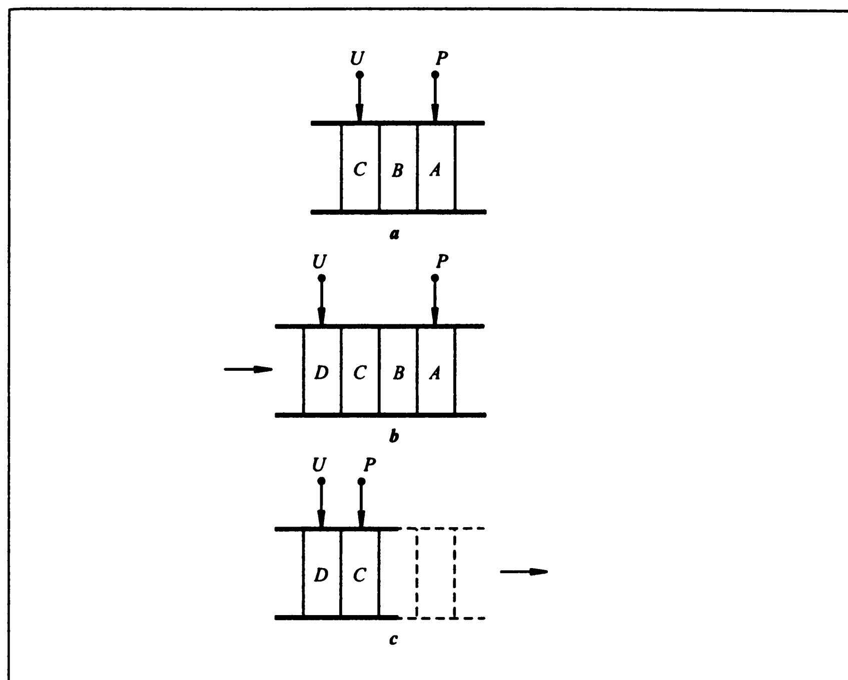


Fig. 6.12. Introducerea şi extragerea elementelor din coadă: *a* – coada iniţială; *b* – introducerea elementului D; *c* – extragerea elementelor A, B

Extragerea unui element din coadă (*fig. 6.12*) este efectuată de secvenţa:

```
R:=P; { memorarea adresei primei celule }
{prelucrarea informaţiei din cîmpul R^.Info }
P:=P^.Urm; {eliminarea primei celule }
dispose(R); {distrugerea celulei extrase }
```

Exemplu:

```
Program P128;
{Crearea şi prelucrarea unei cozi }
type AdresaCelula=^Celula;
     Celula=record
              Info : string;
              Urm : AdresaCelula;
```

```
            end;
var P,                    { adresa primului element }
    U : AdresaCelula; { adresa ultimului element }
    c : char;

procedure Introduc;
var R : AdresaCelula;
begin
 new(R);
 write('Daţi elementul ce urmează');
 writeln(' să fie introdus:');
 readln(R^.Info);
 R^.Urm:=nil;
 if P=nil then begin P:=R; U:=R end
          else begin U^.Urm:=R; U:=R end;
end; { Introduc }

procedure Extrag;
var R : AdresaCelula;
begin
 if P=nil then writeln('Coada este vidă')
          else begin
                 R:=P;
                 write('Este extras');
                 writeln(' elementul:');
                 writeln(R^.Info);
                 P:=P^.Urm;
                 dispose(R);
               end;
end; { Extrag }
procedure Afis;
var R : AdresaCelula;
begin
 if P=nil then writeln('Coada este vidă')
          else begin
                 write('Coada include');
                 writeln(' elementele:');
                 R:=P;
                 while R<>nil do
                  begin
                    writeln(R^.Info);
                    R:=R^.Urm;
                  end;
               end;
 readln;
end; { Afis }

begin
 P:=nil; U:=nil; { iniţial coada este vidă }
 repeat
```

```
  writeln('Meniu:');
  writeln('I - Introducerea elementului;');
  writeln('E - Extragerea elementului;');
  writeln('A - Afişarea cozii la ecran;');
  writeln('O - Oprirea programului');
  write('Opţiunea='); readln(c);
  case c of
      'I' : Introduc;
      'E' : Extrag;
      'A' : Afis;
      'O' :
      else writeln('Opţiune necunoscută')
  end;
 until c='O';
end.
```

Cozile mai poartă numele de **liste** ***FIFO*** (*first in, first out* — primul element intrat în coadă va fi primul ieşit din coadă). Menţionăm că simularea cozilor cu ajutorul tablourilor unidimensionale este ineficientă din cauza migrării elementelor cozii spre ultima componentă a tabloului.

Întrebări şi exerciţii

❶ Pot fi oare folosite listele bidirecţionale pentru crearea unei *cozi*? Care ar fi avantajele unei astfel de structuri de date?

❷ Elaboraţi o funcţie care returnează numărul elementelor unei cozi.

❸ Avioanele care solicită aterizarea pe o anumită pistă a unui aeroport formează un fir de aşteptare. Elaboraţi un program care citeşte de la tastatură şi afişează pe ecran datele despre fiecare avion care solicită aterizarea şi avionul care aterizează. Datele în studiu includ:

- numărul de înmatriculare (`integer`);
- tipul avionului (`string`);
- numărul rutei (`integer`).

❹ Prin **coadă cu priorităţi** vom înţelege o coadă în care elementul de introdus se inserează nu după ultimul element al cozii, ci înaintea tuturor elementelor cu o prioritate mai mică. Priorităţile elementelor se indică prin numere întregi. Elalaboraţi un program care:

a) creează o coadă cu priorităţi;
b) introduce în coadă elementele specificate de utilizator;
c) extrage elementele din coadă;
d) afişează coada cu priorităţi pe ecran.

❺ Se consideră liste bidirecţionale care îmbină proprietăţile stivei şi cozii: introducerile şi extragerile se fac la ambele capete. Elaboraţi un program care efectuează următoarele operaţii:

a) creare;
b) adăugare la dreapta;

c) adăugare la stînga;
d) ştergere la dreapta listei;
e) ştergere la stînga listei;
f) parcurgerea listei de la stînga la dreapta;
g) parcurgerea listei de la dreapta la stînga.

6.8. ARBORI BINARI

Prin **nod** se înţelege o variabilă dinamică de tipul **record** care conţine un cîmp destinat memorării informaţiilor utile şi doi indicatori de adresă.

Arborele binar se defineşte recursiv după cum urmează:

a) un nod este un arbore binar;

b) un nod ce conţine legături către alţi doi arbori binari este un arbore binar.

Prin convenţie, **arborele vid** nu conţine nici un nod. Pentru exemplificare, în *fig. 6.13* este prezentat un arbore binar nodurile căruia conţin informaţia utilă A, B, C, D, E, F, G, H, I, J. Datele necesare pentru crearea şi prelucrarea unui arbore binar pot fi definite prin declaraţii de forma:

```
type  AdresaNod=^Nod;
      Nod=record
            Info : string;
            Stg, Dr : AdresaNod
          end;
var   T : AdresaNod;
```

Pentru a sublinia faptul că arborii binari sînt **structuri recursive** de date, declaraţiile în studiu pot fi transcrise în forma:

```
type  Arbore=^Nod;
      Nod=record
            Info : string;
            Stg, Dr : Arbore;
          end;
var   T : Arbore;
```

Nodul spre care nu este îndreptată nici o legătură se numeşte **rădăcină**. Adresa rădăcinii se memorează în variabila de tip referinţă T. În cazul unui arbore vid T=**nil**.

Cei doi arbori conectaţi la rădăcină se numesc **subarborele stîng** şi **subarborele drept**. Adresa subarborelui stîng se memorează în cîmpul Stg, iar adresa subarborelui drept — în cîmpul Dr.

Nivelul unui nod este, prin convenţie, 0 pentru nodul-rădăcină şi $i + 1$, pentru nodul conectat la un nod de nivelul i. În mod obişnuit, în

reprezentarea grafică a unui arbore binar nodurile se desenează pe niveluri: rădăcina se află pe nivelul 0, vîrfurile conectate la rădăcină — pe nivelul 1 ş.a.m.d. (*fig. 6.13*). Nodurile de pe nivelul $i + 1$, conectate la un nod de pe nivelul i, se numesc **descendenţii** acestuia. În *fig. 6.13* nodul B este descendentul stîng, iar nodul C este descenden-

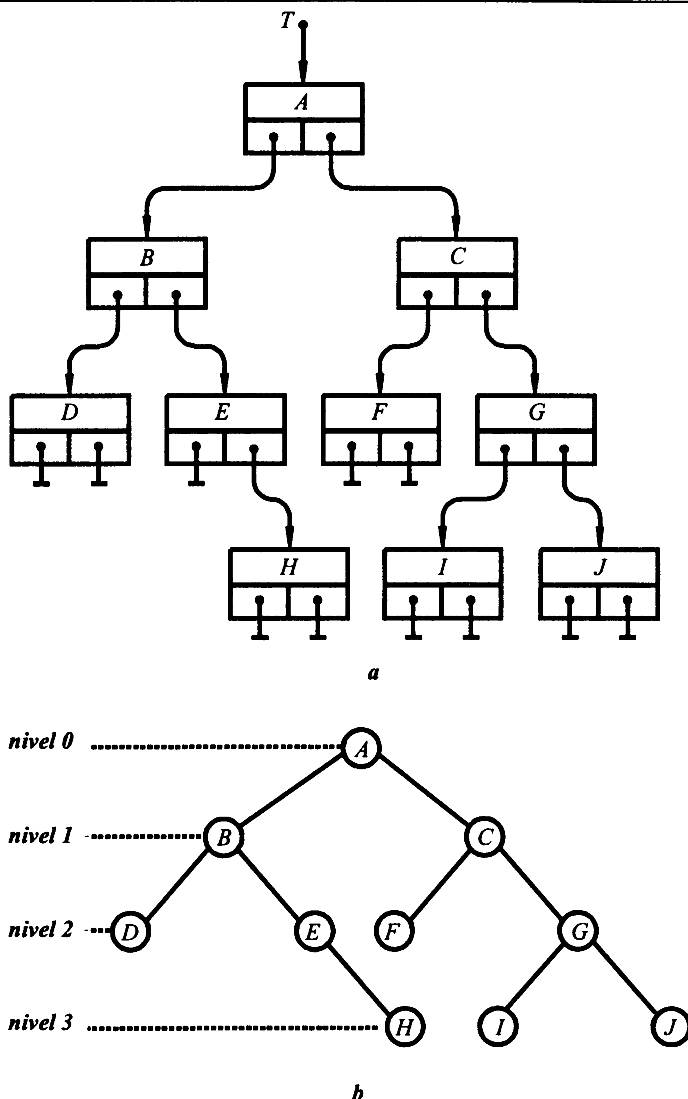


Fig. 6.13. Arbore binar:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

tul drept al nodului A; nodul D este descendentul stîng, iar nodul E — descendentul drept al nodului B ş.a.m.d.

Dacă un nod *x* este descendentul altui nod *y*, îl numim pe acesta din urmă **părintele** nodului *x*. În *fig. 6.13* nodul A este părintele nodurilor B şi C; nodul B este părintele nodurilor D şi E ş.a.m.d.

Un nod la care nu este conectat nici un subarbore este un **nod terminal**. În caz contrar nodul este **neterminal**. Prin **înălţimea** arborelui binar înţelegem numărul de nivel maxim asociat nodurilor terminale. Arborele din *fig. 6.13* are înălţimea 3; nodurile D, H, F, I şi J sînt noduri terminale; nodurile A, B, C, E şi G sînt noduri neterminale.

Arborii binari pot fi construiţi în memoria calculatorului cu ajutorul algoritmilor iterativi sau algoritmilor recursivi.

Algoritmul iterativ creează nodurile în ordinea apariţiei lor pe niveluri:

– se creează nodul-rădăcină;
– nodul-rădăcină se introduce într-o coadă;
– pentru fiecare nod extras din coadă se creează, dacă există, descendentul stîng şi descendentul drept;
– nodurile nou-create se introduc în coadă;
– procesul de construire a arborelui se încheie cînd coada devine vidă.

Nodurile arborelui din *fig. 6.13* vor fi create de algoritmul iterativ în ordinea: A, B, C, D, E, F, G, H, I, J.

Un algoritm similar poate fi utilizat pentru parcurgerea arborelui binar şi afişarea nodurilor respective pe ecran:

– se creează o coadă care conţine un singur element – nodul-rădăcină;
– fiecare nod extras din coadă este afişat pe ecran;
– descendenţii nodului extras se introduc în coadă;
– procesul de afişare se încheie cînd coada devine vidă.

Exemplu:

```
Program P129;
 { Crearea unui arbore binar - iteraţie }
 type AdresaNod=^Nod;
      Nod=record
                Info : string;
             Stg, Dr : AdresaNod
          end;

      AdresaCelula=^Celula;
      Celula=record
                   Info : AdresaNod;
                   Urm : AdresaCelula
             end;
```

```
var  T : AdresaNod;     { rădăcina }
   Prim,                { primul element din coadă }
  Ultim : AdresaCelula; { ultimul element din coadă }

procedure IntroduInCoada(Q : AdresaNod);
var R : AdresaCelula;
begin
 new(R);
 R^.Info:=Q;
 R^.Urm:=nil;
 if Prim=nil then begin Prim:=R; Ultim:=R end
          else begin Ultim^.Urm:=R; Ultim:=R end;
end; { IntroduInCoada}

procedure ExtrageDinCoada(var Q : AdresaNod);
var R : AdresaCelula;
begin
 if Prim=nil then writeln('Coada este vidă')
          else begin
                 R:=Prim;
                 Q:=R^.Info;
                 Prim:=Prim^.Urm;
                 dispose(R);
               end;
end; { ExtrageDinCoada }
procedure CreareArboreBinar;
var R, Q : AdresaNod;
       s : string;
begin
 T:=nil;                 { iniţial arborele este vid }
 Prim:=nil; Ultim:=nil; { iniţial coada este vidă }
 writeln('Daţi rădăcina:'); readln(s);
 if s<>'' then
     begin
     new(R); { crearea rădăcinii }
     R^.Info:=s;
     T:=R;   { iniţializarea adresei rădăcinii }
     IntroduInCoada(T);
    end;
 while Prim<>nil do  { cît coada nu e vidă }
    begin
     ExtrageDinCoada(R);
     writeln('Daţi descendenţii nodului', R^.Info);
     write('  stîng:  '); readln(s);
     if s='' then R^.Stg:=nil
        else
            begin
             new(Q); R^.Stg:=Q;
             Q^.Info:=s;
```

```
                IntroduInCoada(Q);
              end; { else }
       write(' drept: '); readln(s);
       if s='' then R^.Dr:=nil
          else
              begin
               new(Q); R^.Dr:=Q;
               Q^.Info:=s;
               IntroduInCoada(Q);
              end; { else }
      end; { while }
end; { CreareArboreBinar }
procedure AfisareArboreBinar;
var R : AdresaNod;
begin
 if T=nil then writeln('Arbore vid')
    else
     begin
      writeln('Arborele este format din:');
      Prim:=nil; Ultim:=nil;
      IntroduInCoada(T);
      while Prim<>nil do
            begin
             ExtrageDinCoada(R);
             writeln('Nodul ', R^.Info);
             write(' descendenţi: ');
             if R^.Stg=nil then write('nil, ')
                else begin
                      write(R^.Stg^.Info, ', ');
                      IntroduInCoada(R^.Stg);
                     end;

             if R^.Dr=nil then writeln('nil')
                else begin
                      writeln(R^.Dr^.Info);
                      IntroduInCoada(R^.Dr);
                     end;
            end; { while }
     end; { else }
 readln;
end; { AfisareArboreBinar }

begin
 CreareArboreBinar;
 AfisareArboreBinar;
end.
```

Informaţia utilă asociată fiecărui nod se citeşte de la tastatură. Absenţa descendentului se semnalează prin apăsarea tastei <*ENTER*> (programul citeşte de la tastatură un şir vid de caractere). Menţionăm

că coada creată de programul P129 nu conţine nodurile propriu-zise, ci adresele acestor noduri.

Algoritmul recursiv construieşte arborii binari urmînd direct definiţia respectivă:

– se creează nodul-rădăcină;
– se construieşte subarborele stîng;
– se construieşte subarborele drept.

Nodurile arborelui binar din *fig. 6.13* vor fi create de algoritmul recursiv în ordinea: A, B, D, E, H, C, F, G, I, J.

Exemplu:

```
Program P130;
 { Crearea unui arbore binar - recursie }
 type Arbore=^Nod;
      Nod=record
                Info : string;
              Stg, Dr : Arbore
          end;

 var  T : Arbore;     { rădăcina }

 function Arb : Arbore;
  { crearea arborelui binar }
 var R : Arbore;
     s : string;
 begin
  readln(s);
  if s='' then Arb:=nil
     else begin
            new(R);
            R^.Info:=s;
            write('Daţi descendentul stîng');
            writeln(' al nodului ', s, ':');
            R^.Stg:=Arb;
            write('Daţi descendentul drept');
            writeln('al nodului ', s, ':');
            R^.Dr:=Arb;
            Arb:=R;
          end;
 end; { Arb }

 procedure AfisArb(T : Arbore; nivel : integer);
  { afisarea arborelui binar }
 var i : integer;
 begin
  if T<>nil then
     begin
      AfisArb(T^.Stg, nivel+1);
```

```
      for i:=1 to nivel do write('    ');
      writeln(T^.Info);
      AfisArb(T^.Dr, nivel+1);
    end;
end; { AfisareArb }

begin
 writeln('Daţi rădăcina:');
 T:=Arb;
 AfisArb(T, 0);
 readln;
end.
```

Funcţia `Arb` citeşte de la tastatură informaţia utilă asociată nodului în curs de creare. Dacă se citeşte un şir vid, nu se creează nici un nod şi funcţia returnează valoarea **nil**. În caz contrar, funcţia creează un nod, înscrie şirul de caractere în cîmpul `Info` şi returnează adresa nodului. În momentul cînd trebuie completate cîmpurile `Stg` (adresa subarborelui stîng) şi `Dr` (adresa subarborelui drept) funcţia se autoapelează, trecînd astfel la construcţia subarborelui respectiv.

Procedura `AfisArb` afişează arborele binar pe ecran. Se afişează subarborele stîng, rădăcina şi apoi subarborele drept. Nivelul fiecărui nod este redat prin inserarea numărului respectiv de spaţii.

Comparînd programele `P129` şi `P130`, se observă că prelucrarea structurilor recursive de date, şi anume, a arborilor binari, este mai naturală şi mai eficientă în cazul utilizării unor algoritmi recursivi.

Arborii binari au numeroase aplicaţii, una dintre cele specifice fiind reprezentarea expresiilor în scopul prelucrării acestora în translatoarele limbajelor de programare.

Întrebări şi exerciţii

❶ Cum se defineşte un *arbore binar*? Explicaţi termenii: *rădăcină, subarborele stîng, subarborele drept, descendent, nivel, nod terminal, nod neterminal, înălţimea arborelui binar.*

❷ Formulaţi algoritmii iterativi destinaţi creării şi afişării arborilor binari.

❸ Cum se construieşte un arbore binar cu ajutorul algoritmului recursiv?

❹ Elaboraţi un program care construieşte arborele genealogic propriu pe parcursul a trei sau patru generaţii. Nodul-rădăcină conţine numele, prenumele şi anul naşterii, iar nodurile descendente conţin datele respective despre părinţi.

❺ Cum trebuie modificată procedura `AfisArb` din programul `P130` pentru ca arborele binar să fie afişat în ordinea: subarborele drept, nodul-rădăcină, subarborele stîng?

❻ Scrieţi o funcţie recursivă care returnează numărul nodurilor unui arbore binar. Transcrieţi această funcţie într-o formă nerecursivă.

❼ Organizarea unui turneu "prin eliminare" este redată cu ajutorul unui arbore binar. Nodurile arborelui în studiu conţin următoarea informaţie:

– numele (**string**);
– prenumele (**string**);
– data naşterii (ziua, luna, anul);
– cetăţenia (**string**).

Fiecărui jucător îi corespunde un nod terminal, iar fiecărui meci – un nod neterminal. În fiecare nod neterminal se înscriu datele despre cîştigătorul meciului la care au participat cei doi jucători din nodurile descendente. Evident, rădăcina arborelui va conţine datele despre cîştigătorul turneului.
Scrieţi un program care creează în memoria calculatorului şi afişează pe ecran arborele unui turneu prin eliminare.
Indicaţie: Se porneşte de la o listă a jucătorilor. Cîştigătorii meciurilor din prima etapă se includ într-o altă listă. În continuare se formează lista cîştigătorilor meciurilor din etapa a doua ş.a.m.d.

❽ Cum trebuie modificată funcţia `Arb` din programul `P130` pentru ca arborele binar să se construiască în ordinea: `A, C, G, J, I, F, B, E, H, D`?

❾ Funcţia `Arb` din programul `P130` construieşte arborii binari în ordinea: nodul-rădăcină, subarborele stîng, subarborele drept. Scrieţi o procedură nerecursivă care construieşte arborii binari în aceeaşi ordine.
Indicaţie: Se utilizează o stivă elementele căreia sînt noduri. Iniţial stiva va conţine numai nodul-rădăcină. Pentru fiecare nod din vîrful stivei se va construi subarborele stîng, iar apoi — subarborele drept. Nodurile nou-create se introduc în stivă. După construirea subarborelui drept nodul respectiv este scos din stivă.

6.9. PARCURGEREA ARBORILOR BINARI

Operaţiile care se pot efectua asupra arborilor binari se împart în două mari categorii:

– operaţii care modifică structura arborelui (inserarea sau eliminarea unui nod);

– operaţii care păstrează intactă structura arborelui (căutarea unei informaţii, tipărirea informaţiilor asociate unui nod etc.). Una din problemele care apar în mod frecvent la efectuarea acestor operaţii este necesitatea de a parcurge sau a traversa arborele binar.

Prin **parcurgerea unui arbore** se înţelege examinarea în mod sistematic a nodurilor sale astfel încît informaţia din fiecare nod să fie prelucrată o singură dată. Există trei modalităţi de parcurgere a arborilor binari: nodurile pot fi vizitate în preordine, inordine şi postordine. Aceste trei metode sînt definite recursiv: dacă arborele este vid, atunci el este parcurs fără a se face nimic; astfel parcurgerea se face în trei etape.

Parcurgerea în preordine sau traversarea *RSD*:

1) se vizează rădăcina;
2) se traversează subarborele stîng;
3) se traversează subarborele drept.

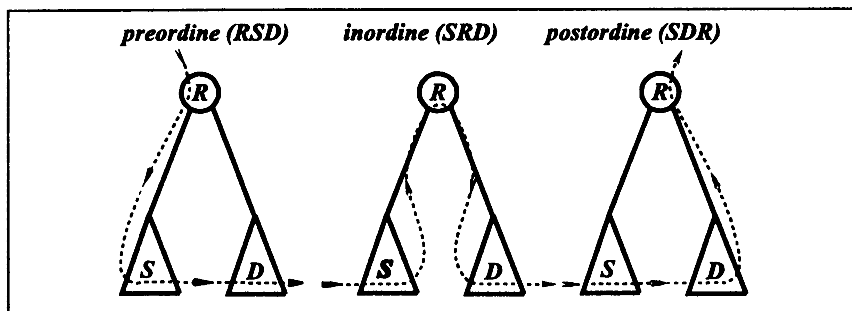


Fig. 6.14. Metode de parcurgere a arborilor binari

Parcurgerea în inordine sau traversarea *SRD*:

1) se traversează subarborele stîng;
2) se vizitează rădăcina;
3) se traversează subarborele drept.

Parcurgerea în postordine sau traversarea *SDR*:

1) se traversează subarborele stîng;
2) se traversează subarborele drept;
3) se vizitează rădăcina.

Notaţiile *RSD, SRD* şi *SDR* reprezintă ordinea în care vor fi vizate rădăcina (*R*), subarborele stîng (*S*) şi subarborele drept (*D*). Metodele de parcurgere a arborilor binari sînt ilustrate în *fig. 6.14*.

Pentru arborele din *fig. 6.13* parcurgerea în preordine furnizează nodurile în ordinea:

A, B, D, E, H, C, F, G, I, J;

parcurgerea în inordine furnizează nodurile în ordinea:

D, B, E, H, A, F, C, I, G, J;

iar parcurgerea în postordine conduce la:

D, H, E, B, F, I, J, G, C, A.

Prezentăm mai jos un program PASCAL de parcurgere a unui arbore binar după toate cele trei metode.

```
Program P131;
 { Parcurgerea arborelui binar }
 type Arbore=^Nod;
      Nod=record
                Info : string;
             Stg, Dr : Arbore
          end;
 var  T : Arbore;    { rădăcina }
```

```
function Arb : Arbore;
 { crearea arborelui binar }
var R : Arbore;
    s : string;
begin
 readln(s);
 if s='' then Arb:=nil
    else begin
          new(R);
          R^.Info:=s;
          write('Daţi descendentul stîng');
          writeln(' al nodului ', s, ':');
          R^.Stg:=Arb;
          write('Daţi descendentul drept');
          writeln(' al nodului ', s, ':');
          R^.Dr:=Arb;
          Arb:=R;
         end;
end; { Arb }

procedure AfisArb(T : Arbore; nivel : integer);
 { afişarea arborelui binar }
var i : integer;
begin
 if T<>nil then
    begin
     AfisArb(T^.Stg, nivel+1);
     for i:=1 to nivel do write('    ');
     writeln(T^.Info);
    AfisArb(T^.Dr, nivel+1);
    end;
end; { AfisareArb }

procedure Preordine(T : Arbore);
 { traversare RSD }
begin
 if T<>nil then begin
                  writeln(T^.Info);
                  Preordine(T^.Stg);
                  Preordine(T^.Dr)
                end;
end; { Preordine }

procedure Inordine(T : Arbore);
 { traversare SRD }
begin
 if T<>nil then begin
                  Inordine(T^.Stg);
                  writeln(T^.Info);
                  Inordine(T^.Dr)
```

```
                    end;
end; { Preordine }

procedure Postordine(T : Arbore);
 { traversare SDR }
begin
 if T<>nil then begin
                  Postordine(T^.Stg);
                  Postordine(T^.Dr);
                  writeln(T^.Info)
                end;
end; { Postordine }

begin
 writeln('Daţi rădăcina:');
 T:=Arb;
 AfisArb(T, 0);
 readln;
 writeln('Parcurgere în preordine:');
 Preordine(T);
 readln;
 writeln('Parcurgere în inordine:');
 Inordine(T);
 readln;
 writeln('Parcurgere în postordine:');
 Postordine(T);
 readln;
end.
```

Menţionăm că funcţia `Arb` creează nodurile, parcurgînd arborele binar în curs de construcţie în preordine. Procedura `AfisArb` afişează nodurile parcurgînd arborele binar în inordine.

Întrebări şi exerciţii

❶ Ce operaţii pot fi efectuate asupra arborilor binari?

❷ Explicaţi metodele de parcurgere a arborilor binari. Daţi exemple.

❸ Scrieţi listele de noduri obţinute în urma celor trei metode de parcurgere a arborelui binar din *fig. 6.15*.

❹ Transcrieţi procedurile `Preordine`, `Inordine` şi `Postordine` din programul `P131` în formă nerecursivă.

❺ Scrieţi un subprogram care returnează înălţimea arborelui binar.

❻ Elaboraţi un program care afişează pe ecran toate nodurile aflate pe un nivel dat într-un arbore binar.

❼ Elaboraţi o procedură recursivă care parcurge un arbore binar în ordinea:

a) *RDS* (rădăcina — subarborele drept — subarborele stîng);
b) *DRS* (subarborele drept — rădăcina — subarborele stîng);
c) *DSR* (subarborele drept — subarborele stîng — rădăcina).

Transcrieţi procedura elaborată într-o formă nerecursivă.

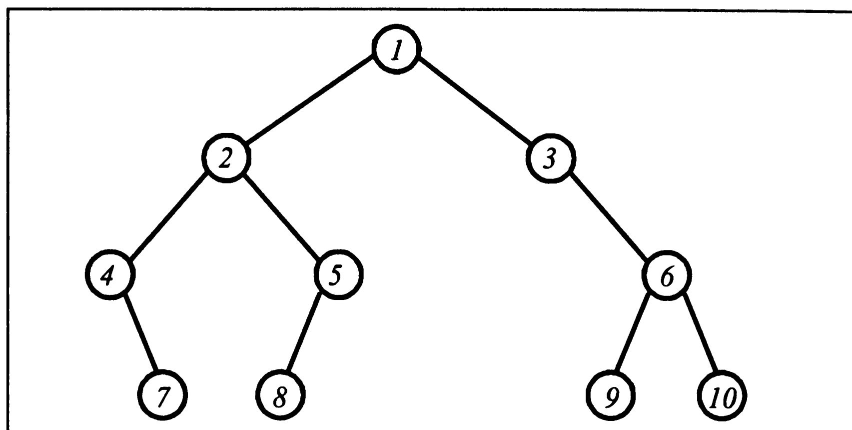


Fig. 6.15. Arbore binar

❽ Scrieţi un subprogram care afişează la ecran nivelul fiecărui nod dintr-un arbore binar.

❾ Se dă un arbore binar în care nodurile terminale reprezintă numere întregi, iar cele neterminale operaţiile binare +, –, *, mod, div. Arborele în studiu poate fi considerat ca o reprezentare a unei expresii aritmetice. Valoarea acestei expresii se calculează efectuînd operaţia din nodul-rădăcină asupra valorilor subexpresiilor reprezentate de subarborele stîng şi subarborele drept.
Scrieţi o funcţie care returnează valoarea expresiilor aritmetice reprezentate prin arbori binari.

❿ Se consideră expresiile aritmetice formate din operanzi şi operatorii binari +, –, *, /. Operanzii sînt variabile numele cărora este format dintr-o singură literă şi constante alcătuite dintr-o cifră. Fiecărei expresii aritmetice i se poate asocia un arbore binar după cum urmează:

a) expresiei aritmetice formate dintr-un singur operand i se asociază un arbore binar format doar din nodul ce conţine operandul respectiv;
b) expresiei aritmetice de forma $E_1 \circ E_2$, unde E_1 *şi* E_2 sînt expresii aritmetice, i se asociază un arbore binar care are în nodul-rădăcină operatorul ∘, ca subarbore stîng arborele asociat expresiei E_1, iar ca subarbore drept arborele asociat expresiei E_2.

Valoarea expresiei se calculează efectuînd operaţia din nodul-rădăcină asupra valorilor subexpresiilor reprezentate de subarborele stîng şi subarborele drept. Scrieţi un program care:

a) construieşte arbori binari asociaţi expresiilor aritmetice citite de la tastatură;
b) evaluează expresiile aritmetice reprezentate prin arborii binari.

Indicaţie. Algoritmul va urma definiţia recursivă a arborelui în studiu. Ca operator curent "∘" se poate desemna orice operator +, – din expresia supusă prelucrării. Operatorii *, / pot fi desemnaţi ca operatori curenţi numai cînd expresia supusă prelucrării nu conţine operatorii +, –.

6.10. ARBORI BINARI DE CĂUTARE

Arborii binari sînt utilizaţi frecvent pentru reprezentarea datelor structurate, componentele cărora pot fi regăsite prin intermediul unei chei. În acest caz fiecare nod conţine un cîmp `Cheie` care identifică informaţia asociată nodului. Cheia unui nod este unică în arbore şi, pentru simplificare, este reprezentată printr-un întreg.

Prin **arbore binar de căutare** se înţelege un arbore binar în care cheia oricărui nod este mai mare decît toate cheile nodurilor din subarborele stîng şi mai mică decît toate cheile nodurilor din subarborele drept.

Din definiţie rezultă că parcurgerea în inordine (subarborele stîng — rădăcina — subarborele drept) a unui arbore binar de căutare va furniza nodurile în ordine crescătoare a cheilor, motiv pentru care astfel de arbori se mai numesc **arbori binari de căutare–sortare.**

Un arbore binar de căutare este prezentat în *fig. 6.16.* Pentru simplificare sînt indicate numai cheile asociate nodurilor respective. Parcurgerea în inordine furnizează nodurile în ordinea: `1, 2, 3, 4, 5, 6, 8, 9`.

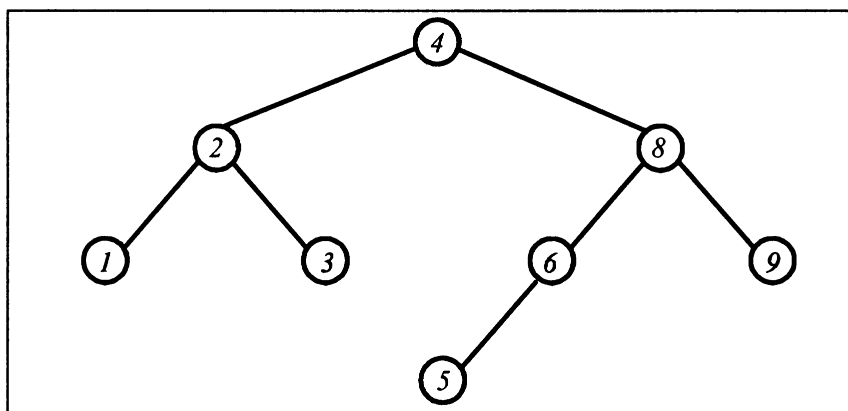


Fig. 6.16. Arbore binar de căutare

Datele necesare pentru crearea şi prelucrarea unui arbore binar de căutare pot fi definite prin declaraţii de forma:

```
type  Arbore = ^Nod;
      Nod = Record
            Cheie : Integer;
            Info : String;
            Stg, Dr : Arbore
            end;
var   T : Arbore;
```

Operaţiile executate frecvent asupra unui arbore binar de căutare sînt căutarea şi inserarea unui nod.

Căutarea nodului ce conţine o anumită cheie se realizează foarte simplu cu ajutorul recursiei:

– dacă nodul-rădăcină conţine cheia necesară, căutarea se termină cu succes;

– dacă nu, în funcţie de rezultatul comparării cheilor respective, căutarea continuă în subarborele stîng sau în subarborele drept.

Dacă subarborele respectiv este vid, atunci căutarea eşuează.

De exemplu, pentru arborele din *fig. 6.16* şi cheia 5, ordinea de vizitare a nodurilor va fi: 4, 8, 6, 5.

Inserarea unui nod ce conţine o anumită cheie se realizează în mod similar:

– dacă arborele este vid, el se substituie cu nodul care urmează să fie inserat;

– în caz contrar, în funcţie de rezultatul comparării cheilor respective, inserarea se va face în subarborele stîng sau subarborele drept.

De exemplu, la inserarea unui nod cu cheia 7 în arborele din *fig. 6.16* ordinea de vizitare a nodurilor va fi: 4, 8, 6. Nodul de inserat va deveni subarborele drept al nodului 6.

Programul care urmează implementează principalele operaţii realizate cu arborii binari de căutare.

```
Program P132;
 { Arbori binari de căutare }
 type Arbore=^Nod;
      Nod=record
                  Cheie : integer;
                   Info : string;
               Stg, Dr : Arbore
          end;
 var T,          { rădăcina }
     P : Arbore;
    ch : integer;
   inf : string;
     c : Char;

 procedure Cautare(T : Arbore; ch : integer;
                 var P : Arbore);
 begin
  if T=nil then P:=nil
     else if ch=T^.Cheie then P:=T
             else if ch<T^.Cheie then
                    Cautare(T^.Stg, ch, P)
                  else
                    Cautare(T^.Dr, ch, P);
```

```
end; { Cautare }

procedure Inserare(var T:Arbore; ch:integer;
                        inf : string);
var Q : Arbore;
begin
 if T=nil then begin
                 new(Q);
                 Q^.Cheie:=ch;
                 Q^.Info:=inf;
                 Q^.Stg:=nil;
                 Q^.Dr:=nil;
                 T:=Q;
               end
    else if ch=T^.Cheie then
             writeln('Nodul există deja')
            else if ch<T^.Cheie
                then Inserare(T^.Stg, ch, inf)
                else Inserare(T^.Dr, ch, inf);
end; { Inserare }
procedure AfisArb(T:Arbore; nivel : integer);
 { afişarea arborelui binar }
var i : integer;
begin
 if T<>nil then
    begin
     AfisArb(T^.Stg, nivel+1);
     for i:=1 to nivel do write('    ');
    writeln(T^.Cheie, '_', T^.Info);
     AfisArb(T^.Dr, nivel+1);
    end;
end; {  AfisareArb }

procedure Inordine(T : Arbore);
 { traversare SRD }
begin
 if T<>nil then begin
                  Inordine(T^.Stg);
                  writeln(T^.Cheie, '_',T^.Info);
                  Inordine(T^.Dr)
                end;
end; { Inordine }

begin
 T:=nil; { iniţial arborele este vid }
 repeat
   writeln('Meniu:');
   writeln('C - Căutare;');
   writeln('I - Inserare;');
   writeln('A - Afişare');
```

```
    writeln('P - Parcurgere în inordine;');
    writeln('O - Oprirea programului.');
    write('Opţiunea='); readln(c);
    case c of
      'C' : begin
              write('Cheia='); readln(ch);
              Cautare(T, ch, P);
              if P=nil then
              writeln('Nod inexistent')
              else
              writeln(P^.Info);
            end;
      'I' : begin
              write('Cheia='); readln(ch);
              write('Info='); readln(inf);
              Inserare(T, ch, inf);
            end;
      'A' : AfisArb(T, 0);
      'P' : Inordine(T);
      'O' :
      else writeln('Opţiune necunoscută');
    end; { case }
  until c='O';
end.
```

Procedura `Cautare` returnează prin parametrul-variabilă `P` adresa nodului ce conţine cheia `ch` specificată de utilizator.

Procedura `Inserare` include în arbore un nod şi înscrie în cîmpurile acestuia cheia `ch` şi informaţia utilă `inf`. Subliniem faptul că adresa `T` a arborelui de căutare se transmite în procedura `Inserare` nu prin parametru-valoare, ci prin parametru-variabilă. Utilizarea parametrului-variabilă este vitală, întrucît substituirea subarborelui vid se realizează prin înlocuirea valorii **`nil`** din unul din cîmpurile de adresă `Stg`, `Dr` ale nodului de pe nivelul precedent cu adresa nodului de inserat.

Avantajul arborilor binari de căutare constă în numărul redus de operaţii de căutare necesare pentru găsirea informaţiei utile asociate unui nod. Evident, în procesul de căutare numărul nodurilor vizitate nu depăşeşte valoarea $h + 1$, unde h este înălţimea arborelui. Menţionăm că în cazul listelor numărul de căutări poate să atingă numărul n de celule din listă. Obişnuit, $h << n$.

Înălţimea arborilor binari de căutare construiţi prin inserări consecutive de noduri depinde de ordinea inserării. În cazul cel mai rău, cînd nodurile se inserează în ordine crescătoare (descrescătoare), arborele degenerează, în fond, într-o listă unidirecţională (*fig. 6.17*). În majoritatea aplicaţiilor se consideră că apariţia unor astfel de cazuri este improbabilă.

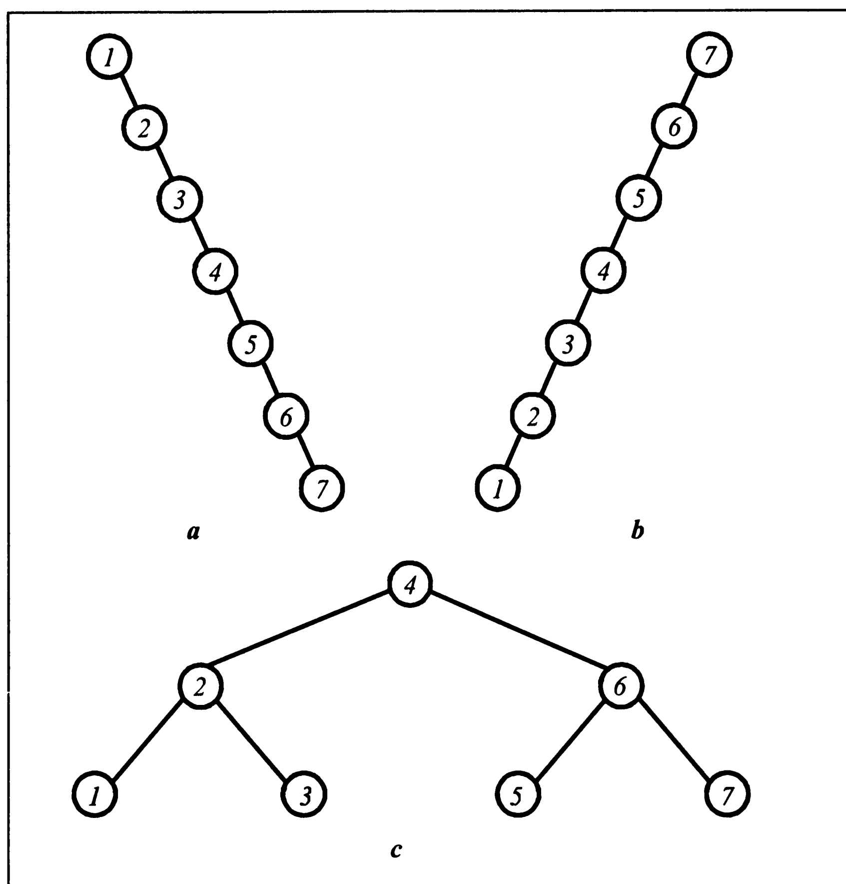


Fig. 6.17. Arbori binari de căutare
construiţi prin inserări consecutive în ordine:
a – crescătoare; *b* – descrescătoare; *c* – definită prin metoda înjumătăţirii

În general, degenerarea arborelui de căutare într-o listă unidirecţională poate fi evitată, inserînd nodurile într-o anumită ordine definită prin **metoda înjumătăţirii**. Conform acestei metode, şirul format din cheile nodurilor de inserat se ordonează crescător (descrescător). Primul se inserează nodul din mijlocul şirului. În continuare se inserează nodurile, cheile cărora se află în mijlocul subşirului stîng şi subşirului drept ş.a.m.d.

De exemplu, în cazul cheilor `1, 2, 3, 4, 5, 6, 7` nodurile pot fi inserate în ordinea `4, 2, 6, 1, 3, 5, 7` (*fig. 6.17*). Se observă că înălţimea arborilor din *fig. 6.17a* şi *6.17b* este $h = 6$, iar înălţimea arborelui din *fig. 6.17c* este $h = 2$.

Întrebări şi exerciţii

❶ Este oare arborele binar din *fig. 6.15* un arbore binar de căutare?

❷ Ce operaţii pot fi executate asupra unui arbore binar de căutare? Cum se execută aceste operaţii?

❸ Transcrieţi procedurile `Inserare` şi `Cautare` din programul `P132` într-o formă nerecursivă.

❹ Într-un registru sînt incluse următoarele date despre fiecare elev: numele şi prenumele, data naşterii (ziua, luna, anul), nota medie. Scrieţi un program care construieşte şi afişează pe ecran arborele binar de căutare ce foloseşte ca cheie:

a) numele şi prenumele;
b) data naşterii;
c) nota medie.

Pentru simplificare se consideră că nu există înregistrări cu chei identice.

❺ **Arborele lexicografic** este un arbore binar de căutare în care cheia reprezintă un cuvînt (o secvenţă nevidă formată din literele alfabetului latin), iar informaţia asociată nodului — numărul de apariţii al cuvîntului respectiv într-un text. Scrieţi un program care construieşte arborele lexicografic al unui fişier text. Se consideră că literele mari şi mici sînt identice. Afişaţi pe ecran arborele lexicografic al programului elaborat.

❻ Arborele binar de căutare este construit prin inserarea consecutivă a *n* noduri. Ordinea de inserare este definită prin metoda înjumătăţirii. Evaluaţi înălţimea arborelui şi numărul de operaţii necesare pentru găsirea unui anumit nod din arbore.

❼ **Arborele echilibrat** este un arbore binar de căutare organizat astfel încît, pentru orice nod din arbore $|h(Stg)-h(Dr)| \leq 1$, unde $h(Stg)$ este înălţimea subarborelui stîng, iar $h(Dr)$ este înălţimea subarborelui drept. Scrieţi un program care:

a) verifică dacă arborele binar de căutare este echilibrat;
b) transformă, dacă e necesar, orice arbore binar de căutare într-un arbore echilibrat.

❽ Pentru a micşora timpul de căutare, informaţia despre fişierele discului magnetic se păstrează în memoria calculatorului în forma unui arbore binar de căutare. Fiecărui fişier îi corespunde un nod care conţine următoarele date:

- numele fişierului;
- data creării (ziua, luna, anul);
- ora creării (ore, minute, secunde);
- lungimea fişierului.

Scrieţi un program care construieşte şi afişează pe ecran arborele binar de căutare ce foloseşte ca cheie:

a) numele fişierului;
b) data şi ora creării.

❾ Scrieţi un subprogram care elimină din arborele binar de căutare nodul ce conţine o anumită cheie. După eliminare arborele rămas trebuie să fie arbore de căutare.
Indicaţie. Se disting următoarele situaţii:

a) nodul de eliminat este un nod terminal — caz în care legătura către acest nod devine `nil`;

b) nodul de eliminat subordonează un singur subarbore — caz în care legătura către nodul în studiu se redirecţionează către subarborele respectiv;

c) nodul de eliminat subordonează doi subarbori — caz în care se va încerca înlocuirea nodului cu cel mai din dreapta nod al subarborelui stîng sau cu cel mai din stînga nod al subarborelui drept.

⑩ Catalogul unei biblioteci este format din fişe. Fiecare fişă include următoarele date:

- numele şi prenumele autorului;
- denumirea cărţii;
- anul apariţiei;
- numărul de inventariere.

Fişele catalogului sînt ordonate alfabetic după numele şi, dacă e necesar, prenumele autorului. Elaboraţi un program care construieşte un arbore binar de căutare şi afişează pe ecran:

a) toate cărţile scrise de un anumit autor;

b) numărul de inventariere al cărţii specificate prin numele autorului, denumirea cărţii şi anul apariţiei.

Programul va asigura includerea şi excluderea anumitor fişe din catalog.

6.11. ARBORI DE ORDINUL *m*

Se consideră variabile dinamice de tipul **`record`** care au în cîmpul legăturilor $m \geq 2$ indicatori de adresă. Ca şi în cazul arborilor binari vom numi astfel de variabile **noduri**.

Arborele de ordinul *m* se defineşte recursiv după cum urmează:

a) un nod este un arbore de ordinul *m*;

b) un nod ce conţine cel mult *m* legături către alţi arbori de ordinul *m* este un arbore de ordinul *m*.

Se consideră că în arbore există cel puţin un nod care subordonează exact *m* subarbori. Prin convenţie, **arborele vid** nu conţine nici un nod.

Arborii de ordinul 2 se numesc **arbori binari** şi au fost studiaţi în paragrafele precedente. Arborii de ordinul 3, 4, 5 ş.a.m.d. se numesc **arbori multicăi** (în engleză *multiway tree*).

Pentru exemplificare, în *fig. 6.18* este prezentat un arbore de ordinul 4. Evident, pentru arborii multicăi termenii rădăcină, subarbore, nivel, părinte, descendent, nod terminal, nod neterminal, înălţime au aceeaşi semnificaţie ca şi pentru arborii binari. Terminologia utilizată în structurile de date în studiu include chiar cuvinte ca *fiu,tată, fraţi,unchi,veri,străbunic* ş.a. cu înţeles similar celui din vorbirea curentă pentru noduri aflate pe diverse niveluri. Într-un limbaj simplist,

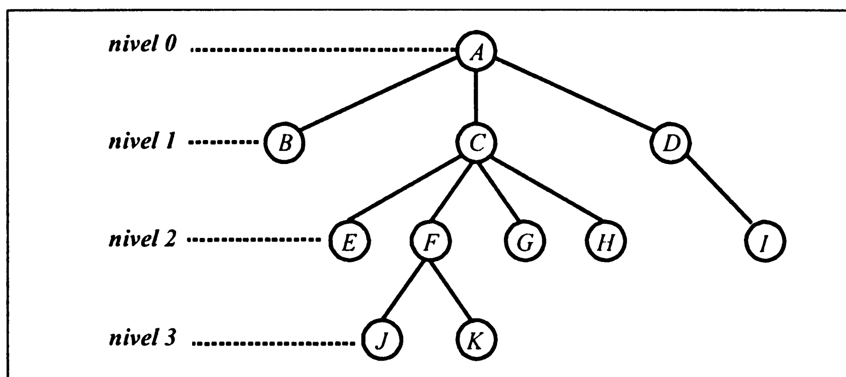


Fig. 6.18. Arbore de ordinul 4

structurile de date în studiu exprimă relaţii de "ramificare" între noduri, asemănătoare configuraţiei arborilor din natură, cu deosebirea că în informatică arborii "cresc" în jos.

Datele necesare pentru crearea şi prelucrarea unui arbore binar de ordinul *m* pot fi definite prin declaraţii de forma:

```
type  Arbore = ^Nod;
      Nod = record;
              Info : string;
              Dsc  : array [1..m] of Arbore
            end;
var   T : Arbore;
```

Adresele descendenţilor unui nod se memorează în componentele `Dsc[1]`, `Dsc[2]`, ..., `Dsc[m]` ale tabloului `Dsc`. Adresa rădăcinii se reţine în variabila de tip referinţă `T`.

Cele mai uzuale metode de parcurgere a arborilor de ordinul *m* sînt parcurgerea în lăţime şi parcurgerea în adîncime.

Parcurgerea în lăţime presupune vizitarea nodurilor în ordinea apariţiei lor pe niveluri. De exemplu, pentru arborele din *fig. 6.18* nodurile vor fi vizitate în ordinea: `A, B, C, D, E, F, G, H, I, J, K`.

În mod obişnuit, parcurgerea în lăţime se realizează cu ajutorul unui algoritm iterativ care utilizează o structură auxiliară de date, şi anume, o coadă formată din adresele nodurilor care vor fi vizitate.

Parcurgerea în adîncime se defineşte recursiv: dacă arborele este vid, el este parcurs fără a se face nimic; altfel se vizitează întîi rădăcina, apoi subarborii de la stînga la dreapta. Pentru arborele din *fig. 6.18* parcurgerea în adîncime furnizează nodurile în ordinea: `A, B, C, E, F, J, K, G, H, D, I`. Parcurgerea în adîncime se realizează foarte simplu cu ajutorul unui algoritm recursiv.

Exemplu:

```
Program P133;
 { Arbori de ordinul m }
 const m=4;
 type Arbore=^Nod;
      Nod=record
                    Info : string;
                    Dsc : array [1..m] of Arbore
          end;

      AdresaCelula=^Celula;
      Celula=record
                    Info : Arbore;
                     Urm : AdresaCelula
             end;

 var  T : Arbore;      { rădăcina }
    Prim,              { primul element din coadă }
  Ultim : AdresaCelula; { ultimul element
                          din coadă }

 procedure IntroduInCoada(Q : Arbore);
 var R : AdresaCelula;
 begin
  new(R);
  R^.Info:=Q;
  R^.Urm:=nil;
  if Prim=nil then begin Prim:=R; Ultim:=R end
           else begin Ultim^.Urm:=R; Ultim:=R end;
 end; { IntroduInCoadă }

 procedure ExtrageDinCoada(var Q : Arbore);
 var R : AdresaCelula;
 begin
  if Prim=nil then writeln('Coada este vidă')
           else begin
                  R:=Prim;
                  Q:=R^.Info;
                  Prim:=Prim^.Urm;
                  dispose(R);
                end;
 end; { ExtrageDinCoadă }

 procedure CreareArbore(var T : Arbore);
 var R, Q : Arbore;
       s : string;
       i : integer;
 begin
  T:=nil;                  { iniţial arborele este vid }
```

```
  Prim:=nil; Ultim:=nil; { iniţial coada este vidă }
  writeln('Daţi rădăcina: '); readln(s);
  if s<>'' then
     begin
       new(R); { crearea rădăcinii }
       R^.Info:=s;
       T:=R;   { iniţializarea adresei rădăcinii }
       IntroduInCoada(T);
     end;
  while Prim<>nil do  { cît coada nu e vidă }
     begin
       ExtrageDinCoada(R);
       for i:=1 to m do R^.Dsc [ i] :=nil;
       writeln('Daţi descendenţii nodului',R^.Info);
       i:=1; readln(s);
       while (i<=m) and (s<>'') do
         begin
           new(Q); R^.Dsc[ i] :=Q; Q^.Info:=s;
           IntroduInCoada(Q);
           i:=i+1; readln(s);
         end;
  end;
end; { CreareArbore }

procedure AfisareArbore(T : Arbore);
var R : Arbore;
    i : integer;
begin
  if T=nil then writeln('Arbore vid')
     else
       begin
         writeln('Arborele este format din:');
         Prim:=nil; Ultim:=nil;
         IntroduInCoada(T);
         while Prim<>nil do
            begin
              ExtrageDinCoada(R);
              writeln('Nodul ', R^.Info);
              write('  Descendenţi: ');
              for i:=1 to m do
                if R^.Dsc[ i] <>nil then
                   begin
                     write(R^.Dsc[ i] ^.Info, ' ');
                     IntroduInCoada(R^.Dsc[ i] );
                   end; { then }
              writeln;
            end; { while }
       end; { else }
  readln;
```

```
end; { AfisareArbore }

procedure InLatime(T : Arbore);
var R : Arbore;
    i : integer;
begin
 if T<>nil then
    begin
      Prim:=nil; Ultim:=nil;
      IntroduInCoada(T);
      while Prim<>nil do
        begin
          ExtrageDinCoada(R);
          writeln(R^.Info);
          for i:=1 to m do
            if R^.Dsc[ i]<>nil then
              IntroduInCoada(R^.Dsc[ i] );
        end; { while }
      end; { then }
end; { InLatime }

procedure InAdincime(T : Arbore);
var i : integer;
begin
 if T<>nil then
    begin
      writeln(T^.Info);
      for i:=1 to m do InAdincime(T^.Dsc[ i] );
    end;
end; { InAdincime }

begin
 CreareArbore(T);
 AfisareArbore(T);
 writeln('Parcurgerea arborelui în lăţime:');
 InLatime(T);
 readln;
 writeln('Parcurgerea arborelui în adîncime:');
 InAdincime(T);
 readln;
end.
```

Informaţia utilă asociată fiecărui nod se citeşte de la tastatură. Absenţa descendenţilor se semnalează prin apăsarea tastei <*ENTER*>. Menţionăm că procedura `CreareArbore` creează nodurile parcurgînd în lăţime arborele în curs de construcţie. Evident, procedura `AfisareArbore` vizitează nodurile în ordinea creării.

Operaţiile frecvent efectuate asupra arborilor multicăi sînt: inserarea sau eliminarea unui nod, căutarea unei informaţii, prelucrarea in-

formaţiilor utile asociate nodurilor ş.a. De obicei, arborii multicăi sînt utilizaţi în cazul aplicaţiilor care implică prelucrarea unor mari cantităţi de date organizate ierarhic pe suporturile externe de informaţie. De exemplu, amintim modul de organizare a discurilor magnetice şi optice în sistemele de operare MS-DOS, UNIX etc. Arborii în studiu sînt de asemenea utilizaţi în aplicaţiile grafice pentru reprezentarea relaţiilor dinamice dintre componentele imaginilor procesate.

Întrebări şi exerciţii

❶ Daţi exemple de arbori de ordinul 3, 5, 6.

❷ Cum se defineşte un arbore de ordinul *m*? Ce operaţii pot fi efectuate asupra arborilor în studiu?

❸ Explicaţi metodele de parcurgere a arborilor multicăi. Daţi exemple.

❹ Scrieţi un program recursiv care construieşte în memoria calculatorului un arbore multicăi. Informaţia utilă asociată nodurilor se citeşte de la tastatură.

❺ Scrieţi o funcţie care returnează:

a) numărul nodurilor unui arbore multicăi;
b) nivelul unui anumit nod din arbore;
c) înălţimea arborelui.

❻ Transcrieţi procedura `InAdincime` din programul `P133` într-o formă nerecursivă.

❼ Cum trebuie modificată procedura `InLatime` din programul `P133` ca nodurile arborelui din *fig. 6.18* să fie vizitate în ordinea: `A, D, C, B, I, H, G, F, E, K, J`?

❽ Cum trebuie modificată procedura `InAdincime` din programul `P133` pentru ca nodurile arborelui din *fig. 6.18* să fie vizitate în ordinea: `A, D, I, C, H, G, F, K, J, E, B`?

❾ Se dă un arbore multicăi, informaţiile din noduri fiind şiruri de caractere. Să se afişeze pe ecran toate şirurile de caractere de lungime pară.

❿ Organizarea datelor de pe discurile magnetice este redată cu ajutorul unui arbore multicăi. Nodurile terminale reprezintă fişierele, iar nodurile neterminale — directoarele. Informaţia utilă asociată fiecărui nod include:

- numele fişierului sau directorului (**string**[8]);
- `extensia` (**string**[3]);
- data şi ora ultimei actualizări (respectiv ziua, luna, anul şi ore, minute, secunde);
- lungimea (`integer`);
- `atribute`le(*'A'*, *'H'*, *'R'*, *'S'*).

Elaboraţi un program care simulează operaţiile de căutare, creare şi ştergere a fişierelor şi directoarelor.

⓫ În unele cazuri ordinul *m* al arborelui multicăi nu este cunoscut în momentul scrierii programului, fapt ce nu permite utilizarea structurilor de date de tipul **array** [1..*m*] **of** `Arbore`. Pentru a depăşi acest inconvenient, tabloul respectiv poate fi înlocuit cu o listă uni- sau bidirecţională.
Elaboraţi subprogramele necesare pentru crearea şi prelucrarea arborilor multicăi de ordin arbitrar.

6.12. TIPUL DE DATE POINTER

Acest paragraf se referă în întregime la implementarea Turbo PASCAL.

Mulţimea de valori ale tipului predefinit de date `pointer` (indicator) constă din adrese şi valoarea specială **`nil`**. Însă, spre deosebire de tipurile de date referinţă adresele cărora identifică numai variabilele dinamice ce aparţin tipului de bază, valorile de tip `pointer` pot identifica variabile dinamice de orice tip. Evident, valoarea **`nil`** nu identifică nici o variabilă dinamică. Prin convenţie, tipul de date `pointer` este **compatibil** cu orice tip de date referinţă.

Operaţiile care se pot face cu valori de tipul de date `pointer` sînt = şi <>. Valorile de acest tip nu pot fi citite de la tastatură şi afişate pe ecran.

O variabilă de tip `pointer` se introduce printr-o declaraţie de forma:

```
var p : pointer;
```

Întrucît astfel de declaraţii nu conţin informaţii despre tipul de bază, tipul variabilei dinamice `p^` este necunoscut. Prin urmare, variabilele de tip `pointer` nu pot fi dereperate, iar scrierea caracterului ^ după astfel de variabile constituie o eroare.

Programul ce urmează ilustrează utilizarea variabilelor de tip `pointer` pentru memorarea temporară a valorilor variabilelor de tip referinţă.

```
Program P134;
  { Tipul de date pointer }
 var p : pointer;
  i, j : ^integer;
  x, y : ^real;
  r, s : ^string;
 begin
  {p va identifica o variabilă dinamică de tipul
    integer }
  new(i); i^:=1;
  p:=i;
  new(i); i^:=2;
  j:=p;
  writeln('j^=', j^); { se afişează 1 }
  {p va identifica o variabilă dinamică de tipul
    real }
  new(x); x^:=1;
  p:=x;
  new(x); x^:=2;
  y:=p;
  writeln('y^=', y^); {se afişează 1.0000000000E+00 }
```

```
  {p va identifica o variabilă dinamică de tipul
    string }
  new(r); r^:='AAA';
  p:=r;
  new(r); r^:='BBB';
  s:=p;
  writeln('s^=', s^); { se afişează AAA }
  readln;
end.
```

Domeniul principal de utilizare a variabilelor de tip `pointer` este gestionarea memoriei interne a calculatorului. În Turbo PASCAL alocarea variabilelor dinamice se execută într-o zonă specială a memoriei interne numită **heap** (grămadă). Adresa de început a *heap*-ului, numită **adresa de bază**, este depusă în variabila predefinită de tip `pointer` `HeapOrg`. Variabila de tip `pointer` `HeapPtr` conţine adresa primei locaţii libere, numită **vîrful** *heap*-ului (*fig. 6.19*).

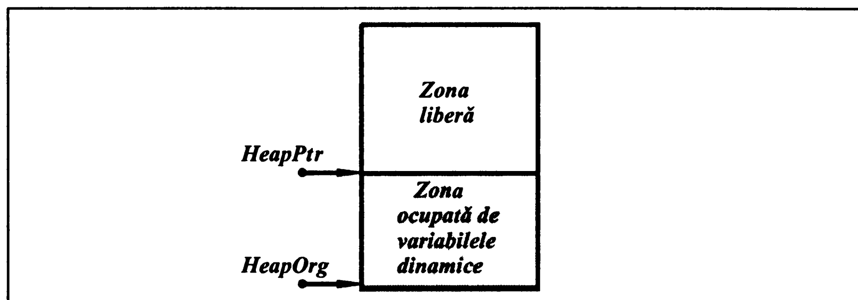


Fig. 6.19. Structura *heap*-ului

Variabilele dinamice sînt create şi depuse în *heap* de procedura `new`. Ori de cîte ori în vîrful *heap*-ului se creează o variabilă dinamică conţinutul variabilei `HeapPtr` este actualizat: valoarea curentă este incrementată cu dimensiunea spaţiului de memorie necesar variabilei dinamice.

Memoria ocupată în *heap* de o variabilă dinamică se eliberează printr-un apel al procedurii `dispose`. Dimensiunea spaţiului ce se eliberează depinde de tipul variabilei dinamice.

Ordinea de apelare a procedurii `dispose` nu coincide în general cu ordinea creării variabilelor dinamice de către procedura `new`. În consecinţă, în *heap* pot apărea goluri. Golurile apărute pot fi refolosite de procedura `new`, dacă variabila dinamică în curs de creare "încape" în spaţiul respectiv.

Eliberarea memoriei ocupate de o structură dinamică de date poate fi efectuată apelînd procedura `dispose` pentru fiecare componentă. Întrucît în program sînt cunoscute numai adresele componentelor pri-

vilegiate, de regulă baza şi vîrful listei, rădăcina arborelui etc., căutarea celorlalte componente cade în sarcina programatorului. Mai mult decît atît, ordinea de apelare a procedurii `dispose` trebuie să asigure păstrarea legăturilor către componentele care încă nu au fost distruse. În caz contrar, componentele respective nu mai sînt referite de nici un indicator de adresă şi devin inaccesibile. Prin urmare, utilizarea procedurii `dispose` pentru eliberarea memoriei ocupate de structuri complexe de date este greoaie şi ineficientă. Acest inconvenient poate fi depăşit cu ajutorul procedurilor predefinite `mark` şi `release`.

Apelul procedurii `mark` are forma:

```
mark(p)
```

unde `p` este o variabilă de tip `pointer`. Procedura memorează adresa vîrfului din `HeapPtr` în variabila `p`.

Apelul procedurii `release` are forma:

```
release(p)
```

Această procedură reface adresa vîrfului în starea înregistrată anterior cu procedura `mark`: valoarea conţinută în variabila de tip `pointer` `p` este depusă în indicatorul `HeapPtr`.

Zona de memorie destinată alocării variabilelor dinamice poate fi gestionată cu ajutorul algoritmului ce urmează:

1) se memorează adresa vîrfului cu procedura `mark`;
2) se creează variabilele dinamice cu procedura `new`;
3) se utilizează variabilele dinamice create;
4) cînd variabilele dinamice nu mai sînt necesare, spaţiul ocupat din *heap* este eliberat cu procedura `release`.

Exemplu:

Se consideră următoarele declaraţii:

```
var i, j, k, m, n : ^integer;
    p : pointer;
```

Să presupunem că sînt executate instrucţiunile:

```
new(i); i^:=1;
new(j); j^:=2;
mark(p);
new(k); k^:=3;
new(m); m^:=4;
new(n); n^:=5;
```

Starea *heap*-ului este prezentată în *fig.6.20a.* Instrucţiunea `mark(p)` a memorat în variabila de tip `pointer` `p` valoarea actuală din `HeapPtr` înainte de crearea variabilei dinamice `k^`.

Dacă acum se execută instrucţiunea

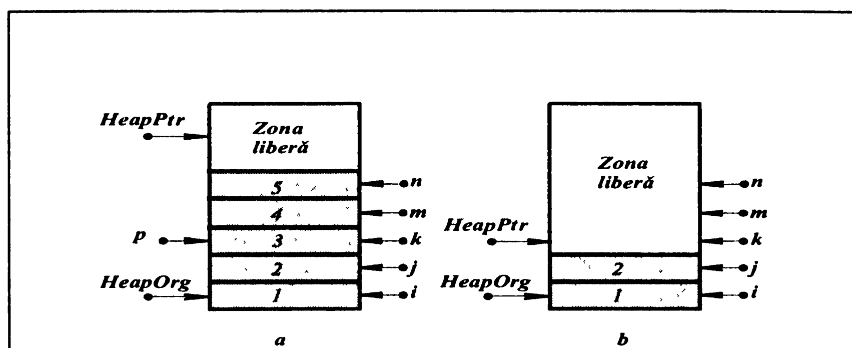


Fig. 6.20. Starea *heap*-ului
pînă (*a*) şi după (*b*) executarea instrucţiunii `release(p)`

```
release(p)
```

memoria ocupată de variabilele dinamice create după apelul procedurii `mark`, şi anume, `k^`, `m^` şi `n^`, va fi eliberată (*fig. 6.20b*).

Deoarece variabila predefinită `HeapOrg` reţine adresa de bază a *heap*-ului, tot spaţiul destinat alocării variabileler dinamice poate fi eliberat cu ajutorul instucţiunii

```
release(HeapOrg)
```

Programul ce urmează ilustrează utilizarea procedurilor `mark` şi `release`.

```
Program P135;
  { Gestionarea memoriei interne }
 type Lista=^Celula;
      Celula=record
                    Info : string;
                     Urm : Lista
             end;
      Stiva=Lista;
      Arbore=^Nod;
      Nod=record
                Info : string;
             Stg, Dr : Arbore
           end;
 var L : Lista;
     S : Stiva;
     T : Arbore;
     p : pointer;

 function Lst : Lista;
 { crearea listei unidirecţionale }
```

```
var R : Lista;
    s : string;
begin
 write('Info='); readln(s);
 if s='' then Lst:=nil
    else
        begin
          new(R);
          R^.Info:=s;
          R^.Urm:=Lst;
          Lst:=R;
        end;
end; { Lst }

procedure AfisLst(L : Lista);
 { afişarea listei }
begin
 if L<>nil then
    begin
      writeln(L^.Info);
      AfisLst(L^.Urm);
    end;
end; { AfisLst }

procedure Stv(var S : Stiva);
 { crearea unei stive }
var R : Stiva;
   st : string;
begin
 S:=nil;
 write('Info='); readln(st);
 while st<>'' do
       begin
         new(R);
         R^.Info:=st;
         R^.Urm:=S;
         S:=R;
         write('Info='); readln(st);
       end;
end; {  Stv }

function Arb : Arbore;
 { crearea arborelui binar }
var R : Arbore;
    s : string;
begin
 readln(s);
 if s='' then Arb:=nil
    else begin
          new(R);
```

```
            R^.Info:=s;
            write('Daţi descendentul stîng');
            writeln(' al nodului ', s, ':');
            R^.Stg:=Arb;
            write('Daţi descendentul drept');
            writeln(' al nodului ', s, ':');
            R^.Dr:=Arb;
            Arb:=R;
           end;
end; { Arb }

procedure AfisArb(T : Arbore; nivel : integer);
 { afişarea arborelui binar }
var i : integer;
begin
 if T<>nil then
    begin
     AfisArb(T^.Stg, nivel+1);
     for i:=1 to nivel do write('    ');
     writeln(T^.Info);
     AfisArb(T^.Dr, nivel+1);
    end;
end; {AfisArb }
 begin
 writeln('Daţi lista:');
 L:=Lst;
 writeln('Lista creată:');
 AfisLst(L);
 mark(p);    { p reţine adresa din HeapPtr }
 writeln('Daţi rădăcina:');
 T:=Arb;
 writeln('Arborele creat:');
 AfisArb(T, 0);
 release(p);{ eliberarea memoriei ocupate de arbore }
 writeln('Daţi stiva:');
 Stv(S);
 writeln('Stiva creată');
 AfisLst(S);
 release(HeapOrg); { eliberarea memoriei ocupate
                     de listă şi stivă }
 readln;
end.
```

Subliniem faptul că în implementările actuale procedurile `dispose` şi `release` nu atribuie valoarea **`nil`** indicatorilor de adresă variabilele dinamice ale căror au fost distruse (*fig. 6.20b*). Întrucît memoria eliberată este refolosită, atribuirile efectuate asupra variabilelor distruse pot altera valorile variabilelor dinamice nou-create.

Întrebări şi exerciţii

❶ Care este mulţimea de valori ale tipului de date `pointer`? Ce operaţii pot fi efectuate cu aceste valori?
❷ Comentaţi următorul program:

```
Program P136;
  { Eroare }
 var   i : ^integer;
       j, k : integer;
       p : pointer;
 begin
  new(i); i^:=1;
  p:=i;
  new(i); i^:=2;
  j:=i^; k:=p^;
  writeln('j+k=', j+k);
 end.
```

❸ Care este domeniul de utilizare a variabilelor de tip `pointer`?
❹ Este oare *heap*-ul o structură de date de tip stivă? Argumentaţi răspunsul.
❺ Lansaţi în execuţie programele ce urmează. Explicaţi rezultatele afişate pe ecran.

```
Program P137;
 var i, j, k, m, n : ^integer;
                 p : pointer;
 begin
  { crearea variabilelor i^, j^, k^ }
  new(i); new(j); new(k);
  i^:=1; j^:=2; k^:=3;
  p:=j; {p reţine adresa din j }
  { distrugerea variabilei j^ şi crearea variabilei m^}
  dispose(j); new(m); m^:=4;
  j:=p; { refacerea adresei din j }
  writeln('i^=', i^, '  j^=', j^, '  k^=', k^);
  { distrugerea variabilei m^ şi crearea variabilei n^ }
  dispose(m); new(n); n^:=5;
  writeln('i^=', i^, '  j^=', j^, '  k^=', k^);
  readln;
 end.
```

```
Program P138;
 var i, j, k, m : ^integer;
 begin
  { crearea variabilelor i^, j^ }
  new(i); new(j);
  i^:=1; j^:=2;

  { eliberarea memoriei heap-ului }
  release(HeapOrg);
```

```
 { crearea variabilelor k^ şi m^ }
 new(k); new(m);
 k^:=1; m^:=2;
 writeln('k^=', k^, ' m^=', m^);
 i^:=3; j^:=4;
 writeln('k^=', k^, ' m^=', m^);
 readln;
end.
```

❻ Scrieţi o procedură care eliberează memoria ocupată de:

a) o listă unidirecţională;
b) un arbore binar;
c) un arbore multicăi.

Memoria trebuie eliberată apelînd procedura `dispose` pentru fiecare componentă a structurii dinamice de date.

❼ Scrieţi un program în care se creează mai întîi o coadă, iar apoi un arbore multicăi. Spaţiul de memorie eliberat după distrugerea cozii trebuie refolosit pentru alocarea arborelui.

Capitolul 7

METODE DE ELABORARE A PRODUSELOR PROGRAM

7.1. PROGRAMAREA MODULARĂ

Programarea modulară urmăreşte reducerea complexităţii programelor mari prin descompunerea acestora în module.

Modulul este un produs program format din descrieri de date şi subprograme destinate prelucrării acestora. Modulele pot fi scrise independent şi compilate separat înainte de a fi încorporate în programul în curs de elaborare. Menţionăm că pentru program se mai utilizează şi denumirea de **modul principal**.

Limbajul-standard nu prevede mijloace pentru programarea modulară. Se consideră că programele PASCAL sînt entităţi monolit care trebuie compilate împreună cu subprogramele pe care, eventual, le conţin. Acest lucru devine incomod în cazul programelor mari care pot include zeci şi chiar sute de subprograme.

În versiunea Turbo PASCAL modulele sînt implementate prin **unităţi de program**. Forma generală a unei **unităţi de program** este:

```
unit <Nume>;
interface
[ uses <Nume>{ ,<Nume>} ;]
[ <Constante>]
[ <Tipuri>]
[ <Variabile>]
[{ <Antet funcţie>; | <Antet procedură>;}]

implementation
[ uses <Nume> { ,<Nume} ;]
[ <Etichete>]
[ <Constante>]
[ <Tipuri>]
[ <Variabile>]
[ <Subprograme>]
[{ function <Identificator>;
<Corp>; |
 procedure <Identificator>;
```

```
<Corp>;} ]
[ begin
[ <Instrucţiune> { ; <Instrucţiune> }] ]
end.
```

În esenţă, o unitate de program constă din trei secţiuni: de interfaţă, de implementare şi de iniţializare.

Secţiunea de interfaţă începe cu cuvîntul-cheie **interface**. Aici se declară constantele, tipurile, variabilele şi subprogramele exportate de unitate. Aceste elemente pot fi referite de orice modul care utilizează direct sau prin tranzitivitate unitatea respectivă. Menţionăm că în secţiunea de interfaţă apar doar antetele funcţiilor şi procedurilor exportate. Dacă unitatea actuală utilizează alte unităţi, numele acestora sînt specificate în clausa **uses**.

Secţiunea de implementare începe cu cuvîntul-cheie **implementation**. Această secţiune conţine declaraţii locale de etichete, constante, tipuri, variabile şi subprograme. Elementele definite aici sînt „ascunse" şi nu pot fi referite de modulele care utilizează unitatea actuală. După declaraţiile locale urmează corpul procedurilor şi funcţiilor, ale căror antete au fost definite în secţiunea de interfaţă. Fiecare subprogram specificat în interfaţă trebuie să aibă un corp. După cuvîntul-cheie **function** sau **procedure** se scrie doar numele subprogramului. Ca şi în cazul declaraţiilor anticipate, nu este necesară descrierea listei de parametri şi a valorii returnate.

Secţiunea de iniţializare începe, dacă există, cu cuvîntul-cheie **begin**. Secţiunea constă dintr-o secvenţă de instrucţiuni şi serveşte pentru atribuirea valorilor iniţiale variabilelor definite în secţiunea de interfaţă. Dacă un program utilizează mai multe unităţi, execuţia programului este precedată de execuţia secţiunilor de iniţializare în ordinea în care aceste unităţi apar în clausa **uses** din program.

Exemplu:

```
Unit U1;
  { Prelucrarea vectorilor }
  interface

   const nmax=100;
   type Vector=array[ 1..nmax] of real;
   var n : 1..nmax;
   function sum(V : Vector) : real;
   function min(V : Vector) : real;
   function max(V : Vector) : real;
   procedure Citire(var V : Vector);
   procedure Afisare(V : Vector);

  implementation
```

```
var i  : 1..nmax;
    s  : real;

function sum;
begin
     s:=0;
     for i:=1 to n do s:=s+V[i];
     sum:=s;
end; { sum }

function min;
begin
     s:=V[1];
     for i:=2 to n do
  if s>V[i] then s:=V[i];
     min:=s;
end; { min }

function max;
begin
     s:=V[1];
     for i:=2 to n do
  if s<V[i] then s:=V[i];
     max:=s;
end; { max }

procedure Citire;
begin
     for i:=1 to n do readln(V[i]);
end; { Citire }

procedure Afisare;
begin
     for i:=1 to n do writeln(V[i]);
end; { Afisare }

begin
 write('n='); readln(n);
end.
```

Unitatea U1 exportă constanta nmax, tipul Vector, variabila n, funcţiile sum, min, max, procedurile Citire şi Afisare. Valoarea iniţială a variabilei n este citită de la tastatură. Elementele în studiu pot fi referite în orice program ce conţine clausa **uses** U1.

Exemplu:

```
Program P139;
  { Utilizarea unităţii U1 }
 uses U1;
```

```
var A : Vector;
 begin
  writeln('Daţi un vector:');
  Citire(A);
  writeln('Aţi introdus:');
  Afisare(A);
  writeln('sum=', sum(A));
  writeln('min=', min(A));
  writeln('max=', max(A));
  readln;
 end.
```

Domeniile de vizibilitate ale declaraţiilor din unităţile de program se stabilesc conform regulilor ce urmează.

1. Declaraţiile din secţiunea de implementare sînt vizibile numai în unitatea actuală.

2. Declaraţiile din secţiunea de interfaţă sînt vizibile în:

– unitatea actuală;

– modulele care utilizează direct unitatea actuală;

– modulele care utilizează unitatea actuală prin tranzitivitate.

Referirea oricărui identificator *id* declarat într-o unitate *v* utilizată prin tranzitivitate se face prin *v.id*.

3. Dacă unul şi acelaşi identificator este declarat în mai multe module, este luată în consideraţie declaraţia cea mai recentă.

Exemplu:

```
Program P140;
 uses U2;
 var x : integer;
 begin
  x:=4;
  writeln('Programul P140:');
  writeln('n=', U3.n);
  writeln('m=', m);
  writeln('x=', x);
  readln;
 end.
```

```
Unit U2;
interface
 uses U3;
 var m : integer;
     x : real;
 implementation
 begin
  writeln('Unitatea U2:');
  m:=2;
  writeln('   m=', m);
```

```
   x:=3.0;
   writeln('   x=', x);
  end.

Unit U3;
interface
 var n : integer;
implementation
begin
 writeln('Unitatea U3:');
 n:=1;
 writeln('n=', n);
end.
```

În programul P140 unitatea U2 este utilizată direct, iar unitatea U3 prin tranzitivitate. Variabila n din modulul U3 este referită prin U3.n. Identificatorul x apare în declaraţiile **var** x: real din unitatea U2 şi **var** x: integer din programul P140. Compilatorul ia în consideraţie ultima declaraţie.

Unităţile de program se clasifică în unităţile-standard, livrate o dată cu compilatorul *Turbo PASCAL*, şi unităţile scrise de utilizator. În continuare prezentăm o caracteristică succintă a unităţilor-standard frecvent utilizate.

System — conţine toate subprogramele predefinite din Turbo PASCAL. Unitatea în studiu se încorporează automat în toate programele, fără a fi necesară clausa **uses**.

Crt — permite utilizarea funcţiilor şi procedurilor referitoare la lucrul cu ecranul în mod text, precum şi cu tastatura şi difuzorul. Accesibilitatea subprogramelor se realizează prin clausa **uses** crt.

Graph — implementează subprogramele destinate unor prelucrări grafice: definiri de ferestre şi pagini, definiri de culori şi palete, desenarea arcurilor, cercurilor, poligoanelor şi altor figuri, salvarea imaginilor etc. Serviciile unităţii de program pot fi accesate prin clausa **uses** graph.

Printer — asigură redirectarea operaţiilor de scriere în fişierul text cu numele lst la imprimantă. Utilizînd unitatea în studiu, programatorul nu mai trebuie să declare, să deschidă şi să închidă acest fişier. Serviciile unităţii Printer devin accesibile unui program sau unei unităţi de program prin specificarea clausei **uses** printer.

Destinaţia şi modul de utilizare a constantelor, tipurilor de date, variabilelor, funcţiilor şi procedurilor din unităţile-standard este inclusă în ghidurile de utilizare şi sistemele de asistenţă *Turbo PASCAL's Online Help*.

Elaborînd propriile unităţi de program, orice utilizator îşi poate crea biblioteci de subprograme ce descriu algoritmi din diverse

domenii: rezolvarea ecuaţiilor, calcule statistice, procesarea textelor, crearea şi prelucrarea structurilor dinamice de date etc. Divizarea unui program mare în module uşurează activitatea de elaborare a produselor program în echipă. În astfel de cazuri fiecare programator scrie, testează şi documentează cîteva module relativ simple, ceea ce contribuie la îmbunătăţirea produsului program rezultat.

Întrebări şi exerciţii

❶ Care sînt avantajele programării modulare? Prevede oare limbajul-standard mijloace pentru programarea modulară?

❷ Care este forma-standard a unei unităţi de program? Precizaţi structura şi destinaţia secţiunilor de interfaţă, de implementare şi de iniţializare.

❸ Cum se determină domeniile de vizibilitate ale declaraţiilor din unităţile de program?

❹ Precizaţi ce va afişa pe ecran programul ce urmează.

```
Program P141;
uses U4;
var s : string;
begin
  s:='BBB';
  writeln('U5.k=', U5.k);
  writeln('U5.m=', U5.m);
  writeln('U5.s=', U5.s);
  writeln('U4.m=', U4.m);
  writeln('U4.s=', U4.s);
  writeln('m=', m);
  writeln('s=', s);
  readln;
end.
```

```
Unit U4;
interface
uses U5;
var m : real;
    s : char;
implementation
begin
  m:=4.0;
  s:='A';
end.
```

```
Unit U5;
interface
var k, m : integer;
    s : real;
implementation
begin
  k:=1;
  m:=2;
```

```
  s:=3.0;
end.
```

❺ Comentaţi programul:

```
Program P142;
  { Eroare }
uses U6;
begin
  writeln('k=', k);
  writeln('m=', m);
  readln;
end.
```

```
Unit U6;
interface
 var k : integer;
implementation
 var m : integer;
begin
  k:=1;
  m:=2;
end.
```

❻ Completaţi unitatea de program `U1` din paragraful în studiu cu un subprogram care:

a) returnează media aritmetică a componentelor unui vector;
b) aranjează componentele în ordine crescătoare;
c) returnează produsul componentelor unui vector;
d) returnează numărul componentelor pozitive;
e) aranjează componentele în ordine descrescătoare.

❼ Scrieţi o unitate de program care oferă descrieri de date şi subprograme pentru prelucrarea matricelor.

❽ Numerele întregi n, $n \leq 10^{254}$, pot fi reprezentate în calculator prin şiruri formate din caracterele *'+'*, *'–'*, *'0'*, *'1'*, *'2'*, ..., *'9'*. Elaboraţi o unitate de program care conţine funcţiile şi procedurile necesare pentru efectuarea următoarelor operaţii:

a) citirea numerelor de la tastatură;
b) afişarea numerelor pe ecran;
c) `+`, `-`, `*`, `mod`, `div`;
d) calcularea factorialului;
e) citirea şi scrierea numerelor în fişiere secvenţiale.

❾ Şirurile de caractere de lungime n, $n \leq 500$, pot fi reprezentate în programele *Turbo PASCAL* prin variabile de tipul

```
type lungime = 0..500;
     SirDeCaractere = record
                        n: lungime;
                        s: array[1..500] of char
                      end;
```

Elaboraţi o unitate de program care conţine funcţiile şi procedurile necesare pentru efectuarea următoarelor operaţii:

a) citirea şirurilor de la tastatură;
b) afişarea şirurilor pe ecran;
c) concatenarea şirurilor;
d) compararea lexicografică;
e) calcularea lungimii unui şir.

⑩ Elaboraţi o unitate de program pentru prelucrarea:

a) listelor unidirecţionale;
b) listelor bidirecţionale;
c) cozilor;
d) stivelor;
e) arborilor binari;
f) arborilor binari de căutare;
g) arborilor multicăi.

Utilizaţi în acest scop declaraţiile de tipuri, funcţii şi proceduri din capitolul 6.

⑪ Găsiţi în sistemul de asistenţă *Turbo PASCAL's Online Help* descrierea unităţilor-standard instalate pe calculatorul D-voastră. Afişaţi pe ecran textul fiecărei unităţi, determinaţi destinaţia şi modul de utilizare a funcţiilor şi procedurilor respective.

7.2. TESTAREA ŞI DEPANAREA PROGRAMELOR

Un program este **corect** dacă :

a) după lansarea în execuţie procesul de calcul se termină;

b) rezultatele obţinute reprezintă o soluţie a problemei pentru rezolvarea căreia a fost scris programul.

În caz contrar programul conţine **erori.**

Asigurarea corectitudinii unui program presupune execuţia sa pentru fiecare combinaţie posibilă de valori ale datelor de intrare. În majoritatea cazurilor acest lucru este imposibil, deoarece domeniul de valori al datelor de intrare este, practic, infinit, iar soluţiile respective sînt necunoscute.

Testarea este o etapă în elaborarea programelor ce are drept scop eliminarea erorilor. Ea se realizează executînd programul cu anumite seturi de date de intrare numite **date de testare** sau, mai simplu, **teste**. În funcţie de modul de selecţie a datelor de testare, deosebim :

– testarea funcţională sau metoda cutiei negre ;
– testarea structurală sau metoda cutiei transparente.

Amintim că termenul *„cutie neagră"* este folosit pentru un sistem, structura internă a căruia este necunoscută.

În cazul **testării funcţionale** datele de testare sînt astfel concepute, încît să se asigure că fiecare funcţie a programului este pe deplin

realizată. Programul este văzut ca o cutie neagră, a cărei funcţionare este determinată prin introducerea unor date şi analiza rezultatelor obţinute. Selectarea datelor de intrare depinde, în mare măsură, de îndemînarea şi experienţa celui care efectuează testarea. În mod obişnuit, se selectează **valori tipice** şi **valori netipice** din domeniul datelor de intrare.

Exemplu. Se consideră programul P143. Textul programului nu este deocamdată prezentat pentru a sublinia faptul că în metoda testării funcţionale el nu este necesar. Programul realizează următoarele funcţii :

– citeşte de la tastatură un şir de numere reale;
– afişează pe ecran media aritmetică a numerelor pozitive din şir.

Evident, domeniul datelor de intrare este format din şiruri de numere reale.

Datele de testare vor include :

a) valorile netipice:
– şir vid;
– şir ce nu conţine numere pozitive;
– şir ce conţine un singur număr pozitiv;

b) valorile tipice:
– şir cu două numere pozitive;
– şir cu trei sau mai multe numere pozitive.

În testarea structurală testele sînt elaborate examinînd structura programului: declaraţiile de date, proceduri şi funcţii, instrucţiunile simple, instrucţiunile structurate etc. Datele de testare vor asigura:

a) execuţia fiecărei instrucţiuni simple (atribuiri, apeluri de proceduri, salturi **goto**);

b) execuţia fiecărui ciclu **for** de zero, unu şi de mai multe ori;

c) execuţia fiecărei instrucţiuni **if**, **repeat**, **while** pentru valorile true şi false ale expresiilor booleene de control;

d) execuţia fiecărui caz din componenţa instrucţiunilor **case**.

Exemplu. Prezentăm textul programului care calculează media aritmetică a numerelor pozitive dintr-un şir:

```
Program P143;
  { Media numerelor pozitive dintr-un şir }
 var n, k : integer;
     x, s : real;
 begin
  n:=0;
  k:=0;
  s:=0;
  writeln('Daţi un şir de numere reale:');
  while not eof do
    begin
```

```
      readln(x);
      n:=n+1;
      if x>0 then
                            begin
                              k:=k+1;
                              s:=s+x;
                            end;
    end; { while }
  if n=0 then writeln('Şir vid')
     else if k=0 then
        writeln('Şirul nu conţine numere pozitive')
     else writeln('Media=', s/k);
  readln;
end.
```

Testul trebuie să asigure execuţia instrucţiunilor **while** şi **if** pentru valorile `true` şi `false` ale expresiilor booleene **not eof**, `x>0`, `n=0` şi `k=0`. Prin urmare, datele de testare vor include:

– un şir vid (**not eof**=`false`, `n=0`);
– un şir nevid (**not eof**=`true`, `n≠0`);
– un şir ce conţine cel puţin un număr pozitiv ($x > 0$, $k \neq 0$);
– un şir nevid ce nu conţine numere pozitive ($x \leq 0$, $k = 0$).

Metoda cutiei transparente poate fi utilizată independent sau împreună cu metoda cutiei negre pentru a îmbunătăţi un test deja obţinut. De exemplu, în cazul programului `P143` şirul ce conţine cel puţin un număr pozitiv poate fi înlocuit cu trei şiruri ce conţin, respectiv unul, două, trei sau mai multe numere pozitive. Aceste date vor asigura execuţia instrucţiunilor `k:=k+1`, `s:=s+x` şi calcularea expresiei `s/k` pentru valorile tipice şi valorile netipice ale variabilelor `k` şi `s`.

Dificultăţile în aplicarea testării structurale sînt legate de prezenţa instrucţiunilor de decizie (**if**, **case**), a celor iterative (**for**, **while**, **repeat**) sau a celei de transfer (**goto**). Acestea determină apariţia unui număr foarte mare de combinări, în care instrucţiunile de atribuire şi apelurile de proceduri pot fi executate.

Depanarea programului constă în localizarea zonelor din program care au condus la apariţia unei erori, identificarea cauzelor erorii şi corectarea acesteia. Depanarea poate fi făcută static (după executarea programului) şi dinamic (în timpul executării).

În metoda **depanării statice** cauzele erorii se stabilesc, analizînd rezultatele derulării programului şi mesajele sistemului de operare. Pentru a facilita procesul de depanare, în program temporar se includ instrucţiuni care afişează pe ecran valorile intermediare ale variabilelor-cheie.

În metoda **depanării dinamice** localizarea erorilor se face urmărind executarea programului la nivel de instrucţiuni. Implementările

actuale ale limbajului permit efectuarea următoarelor operaţii de depanare dinamică:

- execuţia pas cu pas a programului;
- observarea valorilor unor expresii specificate;
- crearea şi eliminarea unor puncte de suspendare a executării;
- modificarea valorilor unor variabile;
- trasarea apelurilor de funcţii şi proceduri;
- tratarea erorilor de intrare–ieşire, a erorilor de depăşire etc.

Descrierea detaliată a operaţiilor în studiu este inclusă în sistemul de asistenţă *Turbo PASCAL's Online Help*.

Eficienţa depanării depinde de modul în care este scris şi testat programul, calitatea mesajelor de eroare generate de calculator şi tipul erorii. De regulă, un test care semnalează prezenţa unei erori este urmat de alte texte organizate în aşa fel, încît să izoleze eroarea şi să furnizeze informaţii pentru corectarea ei.

S-a constatat că testarea şi depanarea ocupă mai mult de jumătate din perioada de timp necesară realizării unui produs program. Complexitatea acestor procese poate fi redusă prin divizarea programelor mari în subprograme sau module şi aplicarea metodelor programării structurate.

Subliniem faptul că testarea programelor este un mijloc eficient de a **depista erorile**, însă nu şi un mijloc de a demonstra absenţa lor. Cu toate că testarea nu demonstrează corectitudinea programului, ea este deocamdată singura metodă practică de certificare a produselor program. În prezent se elaborează metode de verificare bazate pe demonstrarea formală a corectitudinii programului, însă rezultatele cunoscute în această direcţie nu sînt aplicabile programelor complexe.

Întrebări şi exerciţii

❶ Cînd un program PASCAL este corect? Cum poate fi asigurată corectitudinea unui program?

❷ Cum se selectează datele de intrare în metoda testării funcţionale?

❸ Elaboraţi un test funcţional pentru programul `P124` din paragraful 6.4. Programul realizează următoarele funcţii:

- creează o listă unidirecţională;
- afişează lista pe ecran;
- include un anumit element în listă;
- exclude din listă elementul specificat de utilizator.

❹ Precizaţi funcţiile realizate de programele ce urmează şi elaboraţi testele funcţionale:

a) `P49`, `P53`, `P54` şi `P69` din capitolul 3;

b) `P78`, `P82`, `P85`, `P89` şi `P90` din capitolul 4;

c) `P127`, `P128` şi `P132` din capitolul 6.

❺ Cum se selectează datele de intrare în metoda testării structurale?

❻ Elaboraţi teste structurale pentru programele ce urmează:

a) P49, P53, P60, P66 şi P68 din capitolul 3;
b) P78, P82, P85, P93 şi P94 din capitolul 4;
c) P127, P128 şi P130 din capitolul 6.

❼ Care este diferenţa dintre *depanarea statică* şi *depanarea dinamică*?

❽ Găsiţi în sistemul de asistenţă *Turbo PASCAL's Online Help* descrierea operaţiilor de depanare dinamică. Efectuaţi aceste operaţii pentru programele elaborate de D-voastră.

7.3. ELEMENTE DE PROGRAMARE STRUCTURATĂ

Încă din primii ani de activitate în domeniul prelucrărilor de date s-a constatat că testarea, depanarea şi modificarea programelor necesită un mare volum de muncă. Mai mult decît atît, programele complexe ce conţin sute şi mii de instrucţiuni devin greu accesibile chiar şi pentru autorii lor.

Programarea structurată reprezintă un stil, o manieră de concepere a programelor potrivit unor reguli bine stabilite, bazate pe teorema de structură. Conform **teoremei de structură,** orice algoritm poate fi reprezentat ca o combinaţie a trei structuri de control:

– secvenţa (succesiune de două sau mai multe atributuri şi/sau apeluri de proceduri);
– decizia (**if**... **then**... sau **if**... **then**... **else**...);
– ciclul cu test iniţial (**while**... **do**...).

Programarea structurată admite şi utilizarea altor structuri de control, cum sînt:

– selecţia (**case**... **of**...);
– ciclul cu test final (**repeat**... **until**...);
– ciclul cu contor (**for**... **do**...).

Regulile de bază ale programării structurate sînt:

1. Structura oricărui program sau subprogram va fi concepută ca o combinaţie a structurilor de control admise: secvenţa, decizia, selecţia, ciclul.

2. Structura datelor utilizate în program trebuie să corespundă specificului problemelor rezolvate.

3. Lungimea maximă a unei funcţii sau proceduri este de 50–100 linii. Folosirea variabilelor globale nu este încurajată.

4. Identificatorii folosiţi pentru constante, tipuri, variabile, funcţii, proceduri şi unităţi de program trebuie să fie cît mai sugestivi.

5. Claritatea textului trebuie asigurată prin inserarea comentariilor şi alinierea textului în conformitate cu structura logică şi sintactică a instrucţiunilor.

6. Operaţiile de intrare-ieşire vor fi localizate în subprograme separate. Corectitudinea datelor de intrare se verifică imediat după citirea lor.

7. Încuibarea insturcţiunilor **if** mai mult de trei ori trebuie evitată prin folosirea istrucţiunilor **case**.

Programele obţinute conform regulilor în studiu sînt testabile, clare, ordonate, fără salturi şi reveniri. Menţionăm că, conform teoremei de structură, orice program poate fi scris fără a utiliza instrucţiunea **goto**. Totuşi unii autori admit utilizarea acestei instrucţiuni cu condiţia ca ea să fie folosită la minimum, iar salturile să fie efectuate numai în jos.

Întrebări şi exerciţii

❶ Care este justificarea teoretică a programării structurate?
❷ Precizaţi structurile de control necesare şi suficiente pentru reprezentarea oricărui algoritm.
❸ Formulaţi regulile de bază ale programării structurate.
❹ Care sînt avantajele programării structurate?
❺ Corespund oare programele P124, P130 şi P135 din capitolul 6 regulilor de bază ale programării structurate?
❻ Programul ce urmează afişează pe ecran toate reprezentările posibile ale numărului natural *n* ca sumă de numere naturale consecutive.

```
Program P144;
 var  a,i,l,s,n : integer;
              b : boolean;
 begin
 write('n='); readln(n);
 b:=true;
 for i:=1 to ((n+1) div 2) do
 begin
 a:=i;
 s:=0;
 while (s<n) do
 begin
 s:=s+a;
 a:=a+1;
 end;
 if s=n then
 begin
 b:=false;
 write(n, '=', i);
 for l:=i+1 to (a-1) do write('+',l);
 writeln;
 end;
```

```
end;
if b then writeln('Reprezentări nu există');
readln;
end.
```

De exemplu, pentru $n = 15$ obţinem :

```
15=1+2+3+4+5;
15=4+5+6;
15=7+8.
```

Reprezentările în studiu se calculează prin metoda trierii, examinîndu-se şirurile :
1, 2, 3, ..., k;
2, 3, ..., k;
3, ..., k
ş.a.m.d., unde $k = (n + 1)$ div 2. Termenii 1, 2, 3, ..., k se calculează cu ajutorul relaţiei recurente $a = a + 1$.
Aliniaţi textul programului în conformitate cu structura logică şi sintaxa fiecărei instrucţiuni.

Anexa 1. VOCABULARUL LIMBAJULUI PASCAL

1. <*Literă*> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

2. <*Cifră*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3. <*Simbol special*> ::= + | - | * | / | = | < | > |] | [| , | (|) | : | ; | ^ | . | @ | { | } | $ | # | <= | => | <> | := | .. | <*Cuvînt-cheie*> | <*Simbol echivalent*>

4. <*Simbol echivalent*> ::= (* | *) | (. | .)

5. <*Cuvînt-cheie*>::= **and** | **array** | **begin** | **case** | **const** | **div** | **do** | **downto** | **else** | **end** | **file** | **for** | **function** | **goto** | **if** | **in** | **label** | **mod** | **nil** | **not** | **of** | **or** | **packed** | **procedure** | **program** | **record** | **repeat** | **set** | **then** | **to** | **type** | **until** | **var** | **while** | **with**

6. <*Identificator*> ::= <*Literă*> { <*Literă*> | <*Cifră*> }

7. <*Directivă*> ::= <*Literă*> {<*Literă*> | <*Cifră*>}

8. <*Întreg fără semn*> ::= <*Cifră*> {<*Cifră*>}

9. <*Semn*> ::= + | -

10. <*Număr întreg*> ::= [<*Semn*>] <*Întreg fără semn*>

11. <*Factor scală*> ::= <*Număr întreg*>

12. <*Număr real*> ::= <*Număr întreg*> e <*Factor scală*> | <*Număr întreg*>.<*Întreg fară semn*> [e <*Factor scală*>]

13. <*Număr*> ::= <*Număr întreg*> | <*Număr real*>

14. <*Şir de caractere*> ::= '<*Element şir*> { <*Element şir*> }'

15. <*Element şir*> ::= '' | <*Orice caracter imprimabil*>

16. <*Etichetă*> ::= <*Întreg fără semn*>

17. <*Comentariu*> ::= (* <*Orice secvenţă de caractere şi sfîrşit de linie neconţinînd acolade drepte*> *)

Notă. Simbolurile terminale { şi } din formula (17) sînt redate prin simbolurile echivalente respectiv (* şi *).

Anexa 2. SINTAXA LIMBAJULUI PASCAL

1. <*Program*> ::=
   <*Antet program*>
   <*Corp*> .

2. <*Antet program*> ::=
   Program <*Identificator*>
   [(<*Identificator*> {, <*Identificator*>})];

3. <*Corp*> ::= <*Declaraţii*>
<*Instrucţiune compusă*>

4. <*Declaraţii*> ::= [<*Etichete*>]
[<*Constante*>]
[<*Tipuri*>]
[<*Variabile*>]
[<*Subprograme*>]

5. <*Etichete*> ::= **label** <*Etichetă*> {, <*Etichetă*>};

6. <*Constante*> ::=
const <*Definiţie constantă*>; { <*Definiţie constantă*>;}

7. <*Definiţie constantă*> ::= <*Identificator*> = <*Constantă*>

8. <*Constantă*> ::=
[+ | -] <*Număr fără semn*> |
[+ | -] <*Nume de constantă*> |
<*Şir de caractere*>

9. <*Tipuri*> ::= **type** <*Definiţie tip*>; { <*Definiţie tip*> ; }
10. <*Definiţie tip*> ::= <*Identificator*> = <*Tip*>
11. <*Variabile*> ::= **var** <*Declaraţie variabile*> ;
{<*Declaraţie variabile*>;}
12. <*Declaraţie variabile*> ::=
<*Identificator*> {, <*Identificator*>} : <*Tip*>
13. <*Subprograme*> ::= { <*Funcţie*>; | <*Procedură*>; }
14. <*Tip*> ::= <*Identificator*> | <*Tip enumerare*> |
<*Tip subdomeniu*> | <*Tip tablou*> |
<*Tip articol*> | <*Tip mulţime*> |
<*Tip fişier*> | <*Tip referinţă*>
15. <*Tip enumerare*> ::= (<*Identificator*> {, <*Identificator*>})
16. <*Tip subdomeniu*> ::= <*Constantă*> .. <*Constantă*>
17. <*Tip tablou*> ::= [**packed**] **array** (. <*Tip*> {, <*Tip*> } .)
of <*Tip*>
18. <*Tip mulţime*> ::= [**packed**] **set of** <*Tip*>
19. <*Tip fişier*> ::= [**packed**] **file of** <*Tip*>
20. <*Tip referinţă*> ::= ^<*Tip*>
21. <*Tip articol*> ::= [**packed**] **record** <*Listă cîmpuri*> [;] **end**
22. <*Listă cîmpuri*> ::= <*Parte fixă*> [; <*Parte variante*>] |
<*Parte variante*>
23. <*Parte fixă*> ::= <*Secţiune articol*> { ; <*Secţiune articol*> }
24. <*Secţiune articol*> ::= <*Nume cîmp*> {, <*Nume cîmp*> } :
<*Tip*>
25. <*Parte variante*> ::= **case** [<*Identificator*> :] <*Tip*>
of <*Variantă*> {; <*Variantă*> }

26. <Variantă> ::= <Constantă> {, <Constantă> } : ([<Listă cîmpuri>] [;])

27. <Funcţie> ::= <Antet funcţie> ; <Corp> | <Antet funcţie> ; <Directivă>
| **function** <Identificator>; <Corp>

28. <Antet funcţie> ::= **function** <Identificator>
[<Listă parametri formali>] : <Identificator>

29. <Procedură>::=<Antet procedură> ;<Corp> |
<Antet procedură> ;
<Directivă> |
procedure <Identificator> ;<Corp>

30. <Antet procedură> := **procedure** <Identificator>
[<Listă parametri formali>]

31. <Listă parametri formali> ::=
(<Parametru formal> {; <Parametru formal>})

32. <Parametru formal> ::= [**var**] <Identificator>
{, <Identificator>} : <Identificator> | <Antet funcţie> |
<Antet procedură>

33. <Instrucţiune> ::= [<Etichetă> :] <Instrucţiune neetichetată>

34. <Instrucţiune neetichetată> ::=
<Atribuire> | <Apel procedură> | <Instrucţiune compusă> |
<Instrucţiune **if**> | <Instrucţiune **case**> |
<Instrucţiune **while**> | <Instrucţiune **repeat**> |
<Instrucţiune **for**> | <Instrucţiune **with**> |
<Instrucţiune **goto**> | <Instrucţiune de efect nul>

35. <Atribuire> ::= <Variabilă> := <Expresie> |
<Nume funcţie> := <Expresie>

36. <Apel procedură> ::= <Nume procedură>
[<Listă parametri actuali> | <Listă parametri de ieşire>]

37. <Listă parametri actuali> ::=
(<Parametru actual> {,<Parametru actual>})

38. <Parametru actual> ::= <Expresie> | <Variabilă> |
<Nume funcţie> | <Nume procedură>

39. <Nume funcţie> ::= <Identificator>

40. <Nume procedură> ::= <Identificator>

41. <Listă parametri de ieşire> ::=
(<Parametru de ieşire> { , <Parametru de ieşire> })

42. <Parametru de ieşire> ::=
<Expresie> [:<Expresie> [:<Expresie>]]

43. <Instrucţiune compusă> ::= **begin** <Instrucţiune>
{ ; <Instrucţiune> } **end**

44. <*Instrucţiune* **if**> ::=
if <*Expresie booleană*> **then** <*Instrucţiune*>
[**else** <*Instrucţiune*>]
45. <*Instrucţiune* **case**> ::=
case <*Expresie*> **of** [<*Caz*>{; <*Caz*>}] [;] **end**
46. <*Caz*> ::= <*Constantă*> {, <*Constantă*>} : <*Instrucţiune*>
47. <*Instrucţiune* **while**> ::=
while <*Expresie booleană*> **do** <*Instrucţiune*>
48. <*Instrucţiune* **repeat**> ::= **repeat** <*Instrucţiune*>
{;<*Instrucţiune*>}**until** <*Expresie booleană*>
49. <*Expresie booleană*> ::= <*Expresie*>
50. <*Instrucţiune* **for**> ::=
for <*Variabilă*> ::= <*Expresie*> <*Pas*> <*Expresie*>
do <*Instrucţiune*>
51. <*Pas*> ::= **to** | **downto**
52. <*Instrucţiune* **with**> ::= **with** <*Variabilă*>
{,<*Variabilă*>} **do** <*Instrucţiune*>
53. <*Instrucţiune* **goto**> ::= **goto** <*Etichetă*>
54. <*Instrucţiune de efect nul*> ::=
55. <*Variabilă*> ::= <*Identificator*> | <*Variabilă*> (. <*Expresie*>
{ , <*Expresie*> } .) | <*Variabilă*>.
<*Nume cîmp*> | <*Variabilă*>
56. <*Nume cîmp*> ::= <*Identificator*>
57. <*Expresie*> ::= <*Expresie simplă*>
{<*Operator relaţional*> <*Expresie simplă*>}
58. <*Operator relaţional*> ::= < | <= | = | >= | > | <> | **in**
59. <*Expresie simplă*>::=[+ | –] <*Termen*>
{<*Operator aditiv*> <*Termen*>}
60. <*Operator aditiv*> ::= + | – | **or**
61. <*Termen*> ::= <*Factor*>
{<*Operator multiplicativ*> <*Factor*>}
62. <*Operator multiplicativ*> ::= * | / | **div** | **mod** | **and**
63. <*Factor*> ::= <*Variabilă*> | <*Constantă fără semn*> |
<*Apel funcţie*> | **not** <*Factor*> |
(<*Expresie*>) | <*Constructor mulţime* >
64. <*Apel funcţie*> ::= <*Nume funcţie*>
[<*Listă parametri actuali*>]
65. <*Constantă fără semn*> ::= <*Număr fără semn*> |
<*Şir de caractere*> | <*Identificator*> | **nil**
66. <*Constructor mulţime*> ::= (. [<*Specificare element*>
{ , <*Specificare element*> }] .)
67. <*Specificare element*> ::= <*Expresie*> [.. <*Expresie*>]

Notă. Simbolurile neterminale [şi] din formulele (55) şi (66) sînt redate prin simbolurile echivalente respectiv (. şi .).

BIBLIOGRAFIE

Munteanu F., Ionescu T., Muscă Gh. ş.a., Programarea calculatoarelor. Manual pentru liceele de informatică, clasele X-XII, Editura didactică şi pedagogică, R. A., Bucureşti, 1995.

Boian F., Cioban V., Frenţiu M. ş.a., Programarea PASCAL. Programe ilustrative şi probleme propuse pentru elevi şi studenţi. Editura *Promedia plus computers*, Cluj-Napoca, 1995.

Bălănescu T., Gavrilă Ş., Georgescu H. ş.a., Programarea în limbajele PASCAL şi Turbo PASCAL, vol. 1, 2, Editura Tehnică, Bucureşti, 1992.

Rancea D., Limbajul Turbo PASCAL., vol. 1, 2, Editura *Libris*, Cluj, 1993.

Sorin T., Tehnici de programare. Editura *L&S Informat,* Bucureşti 1996.

Sandor Kovacs, Turbo PASCAL 6.0. Ghid de utilizare, Editura *MicroInformatica*, Cluj-Napoca, 1993.

Cristea V., Dumitru P., Giumale C. ş.a., Dicţionar de informatică. Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1981.

Йенсен К., Вирт Н. Паскаль, Руководство пользователя, М., Editura *Финансы и статистика*, 1989.

Вирт Н., Алгоритмы + Структуры данных = Программы, М., Editura *Мир*, 1985.

Вирт Н., Алгоритмы и структуры данных, М., Editura *Мир*, 1989.